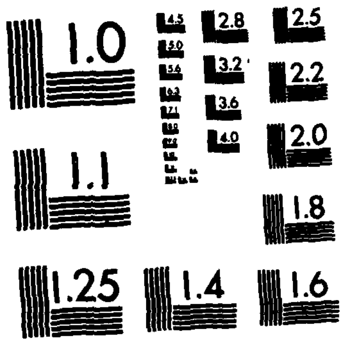
AD-A124 674 ASSOCIATIVE DATA ACCESS METHOD (ADAM)(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING J R HOLTEN DEC 82 AFIT/GCS/EE/82D-19 1/3

UNCLASSIFIED F/G 9/2 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

ASSOCIATIVE DATA ACCESS METHOD (ADAM)

THESIS

AFIT/GCS/EE/82D-19 James R. Holten III
Capt USAF

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY (ATC)
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

①

AFIT/GCS/EE/82D-19

ASSOCIATIVE DATA ACCESS METHOD (ADAM)

THESIS

AFIT/GCS/EE/82D-19 James R. Holten III
Capt USAF

AFIT/GCS/EE/82D-19

ASSOCIATIVE DATA ACCESS METHOD

(ADAM)

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science

by

James R. Holten III, B.S. Mathematics,
B.S. Computer Science

Capt USAF

Graduate Computer Science

December 1982

Approved for Public release; distribution unlimited.

Preface

This work was motivated by the widespread need for rapid multidimensional access to data. Frequent allusions by numerous instructors at the Air Force Institute of Technology and in open literature to associative memories, their high cost, small size, and non-availability, triggered my interest.

This software approach to associative data access is far more flexible than hardware approaches can be, and is far less expensive to use for specific applications. The software developed stores data in an efficient, but unique, manner, and allows rapid multikey access to the data.

It is assumed the reader has a basic knowledge of data structures and some background in relational databases. This report presents the basic approach, the computer program package, and suggestions for further analysis and research.

Thanks are due to my advisor, Dr. Gary B. Lamont, and my readers, LtCol James P. Rutledge, Dr. Henry B. Potoczny, and Dr. Thomas C. Hartrum, for their advice and guidance in the research and preparation of this report.

I would especially like to thank my wife and children for their tolerance, patience, and support during the writing of this thesis.

James R. Holten III

Contents

List of Figures

List of Tables

Abstract

A software solution to the multikey access problem is presented. The result, ADAM, models associative memory techniques to obtain fast retrieval times and efficient data storage. A multidimensional tree structure is used. Each data item key is one dimension, and at each lower level in the tree each dimension is divided into successively smaller half-intervals. Unlike m-way trees with fixed sized nodes and K-D tree levels, each ADAM map level is a linear linked list. Each node of the ADAM level linear linked list is the root of a subtree, or is the terminal node of a data item in the data set. The resulting data structure is, in many cases, more storage efficient than normal linear storage of the data items. This is due to the suppression of duplicate high order bits among the data items. The method allows retrieval of associative data subsets from the associative data set much faster than other multikey access techniques. Analysis of variations on ADAM are suggested, especially for application to very large (over 100000 data items per data set) multiuser databases.

I Introduction

Background

Associative access is often brandished as a panacea for many database retrieval problems. Many people have attempted to implement a good, general purpose method, both in hardware and software (Ref 6;30). All previous attempts have fallen short, each being used for its own application, but none being generally accepted and widely used. This paper discusses software techniques for associative access to data, and derives a new technique.

Noting that no technique is good for all applications, this thesis investigates the performance parameters of several software techniques. The techniques are compared, and selection criteria for matching techniques to applications are suggested. The new software technique is proposed to fill the gaps in performance left by the others.

Associative Access. Associative access is a vague term often replaced by the term "content addressable" (Ref 13; 17; 27; 33). "Content addressable" means accessing data by its value rather than by its location in a data structure. The idea comes from the outward appearance of the human memory retrievals (Ref 35). The human mind seems to associate concepts and thoughts based on their content (Ref 34; 35). A simple event conjures up multitudes of diverse associations in different people's minds. If someone mentions how hot it is, someone else may immediately picture the time they had in the heat at the beach in Florida or on the desert in Arizona. The only relationship between the triggering statement and the resulting thoughts conjured up in people's minds is a concept, or perhaps a single word. Many concept and word associations occur in every mind for most sensory inputs.

Associative data organization in computers is usually a model of this behavior in any way that will give similar outputs for the same inputs (Ref 13; 17; 20; 27; 30). Associative retrieval of information is the driving force behind many data base management systems. An extension to the human memory was needed to hold large quantities of data and give rapid data retrieval (Ref 9). However, lacking the ability to implement a fully content addressable memory on a large enough scale (Ref 30; 33), many approaches have been tried to create a workable substitute. These substitutes include database management systems (Ref 8; 21), associative brain models (Ref 20), and hardware content-scanning associative processors (Ref 27; 30).

When achieved, rapid associative data access can be useful in areas such as pattern recognition, graphic scene analysis (Ref 13), and artificial intelligence (Ref 33). Within these subject areas, associative access can be applied to automated speech recognition, weather prediction, "smart" missile guidance, automated spelling correction, medical diagnosis, aircraft simulator scene generation, and radar target identification. Use in any of these applications, however, requires many retrievals from large data bases, and most of the applications have "tight" retrieval time constraints. Five minutes to analyze a picture can make a data retrieval program unacceptable in many applications. The difference between fast and slow retrieval times can be the access technique used, and to make retrievals fast enough to satisfy the time constraints, associative access has been hailed as the "only way to go" (Ref 17).

Past Methods. Both hardware and software implementations of associative access to data have been created. Most of these implementations have simulated content associative access with combinations of serial and random access techniques. These implementations include serial searches of entire data structures (Ref 27), parallel processors or I/O devices each doing serial searches of large data structures (Ref 30), or linked tree searches of medium to small data structures (Ref 3; 13; 20). The hardware implementations have all been expensive and their expansion capability and access times have limited their usefulness (Ref 27; 30). Most software implementations have been designed to handle specific problems, and their lack of generality precludes their usefulness in other applications (Ref 13; 20).

The Secondary Key Problem. A primary key is any combination of key data fields which can be used to uniquely identify each data item in a data set. When a single key value does not give a unique data item, then the key is referred to as a <u>secondary key</u>. A primary key, unique for each data item, can be used to define an ordering of the data items and used for a binary search. A secondary key may not be unique, and thus an ordering only allows one level of binary search, leaving unordered subsets of the data. Either a new ordering must be created for further binary searching, or a linear search must be used to select the proper elements from among those found satisfying the first search. Knuth (Ref 16:550-567) refers to it as the secondary key problem, and Bentley (Ref 6:397) calls it the "multikey searching problem". The difference is that "multikey" refers to the successive selection on different keys.

Knuth (Ref 16:550-567) discusses retrieval on secondary keys and gives some software solutions. These solutions are inverted files, compound attributes, binary attributes, superimposed coding, combinatorial hashing, generalized tries, and balanced filing schemes. Inverted files are additional orderings for access via multiple key combinations, hereafter referred to as multiple key linear orderings. Compound attributes are orderings which depend on a set of keys, in a specific order of importance. Binary attributes are merely two-valued attributes, allowing combinations of attributes to be represented as a single string of bits. Superimposed coding is similar to binary attributes except that each attribute is given a multibit code, and combinations are represented by "or"ing the multibit codes of all the component attributes together. Combinatorial hashing consists of hashing all the attributes, then concatenating the hashed results to a primary key. Generalized tries (from reTRIEvals) is a variation on the 'trie' search (ref 16:481) and uses an m-way tree search. Balanced filing schemes use each combination of attributes to form an additional inverted file. Of all these methods, the 'trie' search is the closest to the candidate data structures considered for associative access. The candidate structures are the quad tree, the K-D tree, and the CARTAM structure. These three software methods are described in Appendix A, and have been used in associative applications. The quad tree, K-D trees, and CARTAM, and will be discussed further.

Bentley and Finkel (Ref 5) presented the quad tree, a common form of the m-way tree. Since its introduction many others have used it and expanded on the applications of its associative properties (Ref 11; 12; 28). The quad tree is a structure for two dimensional data, and as such its applications have been restricted to such problems as graphic scene representation.

Bentley later introduced the K-D tree (Ref 3) for more general use. The K-D tree has the advantage of being able to represent multidimensional data without changing its basic data structure elements. However, the K-D tree is not optimal, and the user's algorithms must be able to "remember" which dimension is being processed at each node in the structure.

Petersen (Ref 24) created a multidimensional data structure for solving a "near-neighbor" problem application, and called it the Cartesian Access Method, or CARTAM. CARTAM was used in an associative application, but it was not noted as an associative access method. The CARTAM package was created for a specialized application which used Cartesian measures of distance in a multidimensional space, and thus was limited to real spatial coordinates. CARTAM also included a substantial amount of data in each node, making the nodes large and cumbersome.

Summary. Associative access techniques, though sought after by many, are still elusive. Both hardware and software implementations fall short of the performance and flexibility needed to make the techniques generally useable. A variation on the CARTAM structure and algorithms, however, showed promise as a generalized associative access method. All of the candidate approaches have drawbacks, so the secondary key access problem must be analyzed in more depth.

The Problem

The world consists of large volumes of information which living creatures process and store in many diverse ways (Ref 32; 35). To analyze this information and its many facets, a model must be used. Once the model is proposed and accepted, the information, hereafter called data, can be analyzed only to the extent that the model represents the real world, and to the extent that the model can be analyzed. A model must be derived, presented, and analyzed.

Programmers are having to handle projects which involve extremely large amounts of data. Selection of data items from a large data set is easy if there is a single key which is unique for every data item. A binary search may be performed. Bentley (Ref 6:397) points out, however, that multikey retrievals currently give less than satisfactory results. A quicker way to retrieve data using multiple keys must be found.

Software implementation avoids the high cost of developing or purchasing specialized hardware (Ref 17; 27; 30), and can allow the generality needed for widespread acceptance and use of the new associative data structures. A software implementation also can allow the associative access technique to be machine portable.

While serial and random access normally return one data item per retrieval request, associative access is expected to return a set of data elements for each request. Implementing this multiple element retrieval reveals new problems. In what form should the returned set be: a serial table, a random access array, or an associative data structure? How should serial processes and random access data structures interface with the associative data structure? What performance measures should be used to estimate the "goodness" of different associative data structure implementations: update and retrieval times, memory space usage, or complexity of structure? These problems require an analysis of the data relationships and how the relationships are to be used.

The problem is then to implement a generalized associative data access method which has the following properties:

-Models real world associations

-Allows efficient secondary key access to large volumes of data

-Is flexible enough to be implemented on a variety of large and small computers

-Sets up conventions for retrieval of sets of data

-Insures quality design and code through good software engineering techniques.

Also criteria will be listed for choosing which techniques of those analyzed are more efficient for specific applications.

Scope

The CARTAM structure seems to be a promising generalized associative data structure and access method. But there are significant differences in the interpretations of "associative access" which must be resolved. To make the access method useful, not only must it be implemented and evaluated, but the potential users must be shown when the structure can be applied to a problem, how to apply it, and then how to make the best use of the application.

The object is to educate the user, and to give the user a new tool for future software development. This paper presents a possible associative access method, and attempts to give the user a view of data which will make the approach easy to apply. The software implementation of an associative access method gives flexibility far beyond any obtainable currently in hardware, however, it shares some logical concepts with the hardware implementations.

Approach

This paper will present several views of artificial data structures which can be imposed on real world data. Using these views of data, the paper will show how associative access methods can be applied to problems, then an associative data structure and access method will be presented and analyzed.

Current and past literature is surveyed and the applicable techniques of analysis are employed. For analysis purposes, certain notation and terminology has been borrowed and applied to the new model of data and for the description of associativity.

Algorithms to implement an associative access method using the given data structure will be analyzed and compared to other access algorithms. The results will be tabulated.

The model presented is a hierarchical view of data in the real world, and not intended to presume that all data can be represented in a computer. The model presents a logical structure for representing data, and then presents an alternate view of data in terms of types of associations. Using both views, the data structures presented are shown to simulate the real world associations of data.

Using the approach and concepts presented in the model, a candidate associative data structure and access method is proposed, implemented, and its performance analyzed. This proposed data structure is compared to other currently popular data structures which are used for associative retrieval tasks. Analysis is presented for theoretical performance with suggestions for analyzing actual performance.

II <u>Concepts and Requirements</u>

<u>Introduction</u>

This chapter develops concepts and terminology necessary to investigate associativity in data. It then discusses types of data associations and ways to model the associations in a computer. The different association models are developed and their limitations are presented. Methods for further analysis of their usefulness for solving real world problems are presented. An associative model is presented and then ways to access the model are discussed. Finally specific access method requirements are presented.

<u>Basic Concepts</u>

<u>Preliminary Definitions</u>. The terminology and concepts to be used are standardized for the references which follow. First general terms for referring to data are expressed, then more specific terms for data associations are presented.

The structure of real world data can be presented in a hierarchical or recursive structure. All data can be represented in this fashion as simply as stating the words "all data". This can be decomposed at lower levels by such subdivisions as "all data relevant to the problem at hand" and "all data not relevant to the problem at hand."

A <u>data set</u> is any collection of data. The components of the data set are <u>data items</u>. "Data set" and "data item"

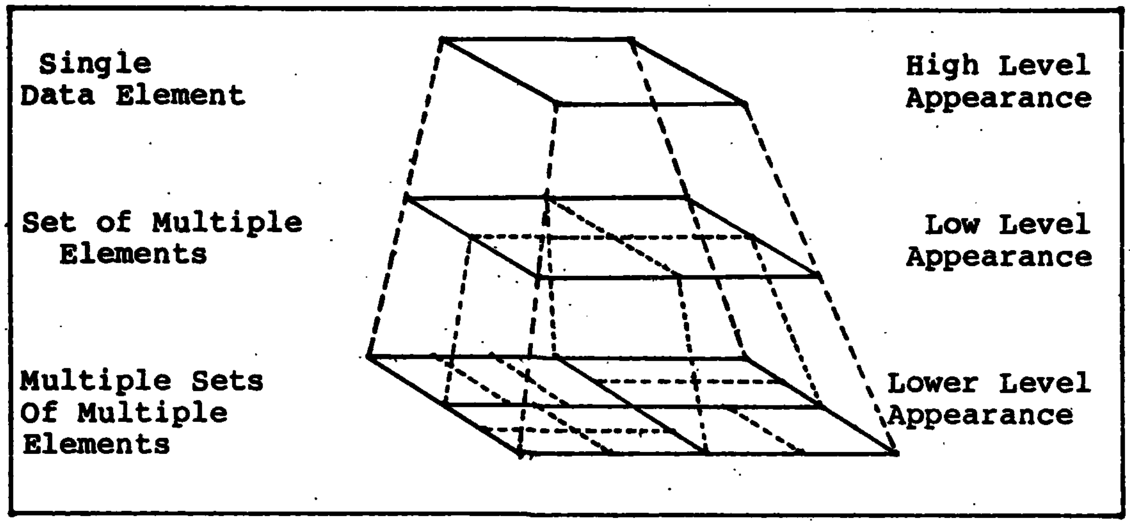


Fig 1. Hierarchy of Data Elements, Levels of Observation

will be used to refer to arbitrary data, while "data element" will refer to structured data components.

A <u>data element</u> is a recursively defined unit of data. It is an arbitrary data set, referenced as a single unit for purposes of clarity of understanding of the underlying data structure and its interrelationships. A <u>fundamental data element</u> is a data element which cannot be further subdivided (atomic). A fundamental data element could be a measured value, a Boolean flag, a character, or a label representing another data element. The makeup of data elements can be shown, using Backus Naur Form (Ref 14:110), as

```
<DE>::= { <CDE> } | <FDE>
<CDE>::= <DE> | <DE> <CDE>
```

where

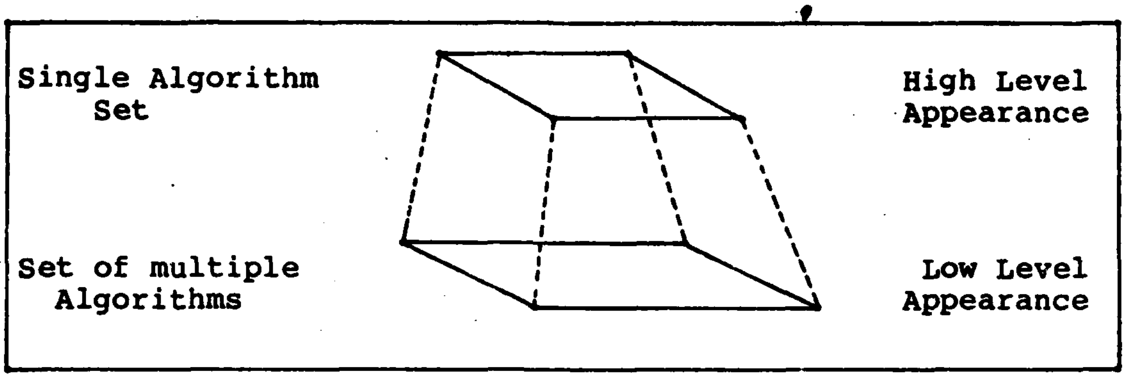


Fig 2. Hierarchy of Algorithm Sets

DE is data element,

CDE is collection of data elements, and

FDE is a fundamental data element.

Data can be thought of as having levels of organization as in Figure 1. Whether a collection of data is considered a single data element or a collection of data elements depends completely on the user's level of observation and abstraction. The data elements at any level may take on data structures classified as simple, compound, or complex, corresponding to different data forms found in the real world and in programming languages as shown in Table I. The classification is affected by not only the data structure, but also the associated algorithm set.

An algorithm set is any collection of algorithms which may be accessed as one single unit. Figure 2 shows how several algorithm sets may be combined into one higher level algorithm set. A collection of integer manipulation

TABLE I

Examples of Data Types

Data Type	Computer Languages			Other Examples	
	HP ATS BASIC .	PASCAL .	FORTRAN IV	Relational Databases	Real World
Simple	Integer, Real, Boolean			A single attribute in a single tuple of a relation, a single tuple, or a single relation.	A label, boolean value, measured value a picture, or a time interval of sound.
	matrix, character string	char, other scalars	double precision		
Compound	Array			All tuples in a relation	All pixels in a T.V. picture sequence of measurements.
	matrix, character string		(vector, matrix)		
Complex	All variables used in a program			All attributes in a single tuple	All the parts in a computer, objects in a picture.
		The parameter string in a subroutine or procedure call			
		record, set			

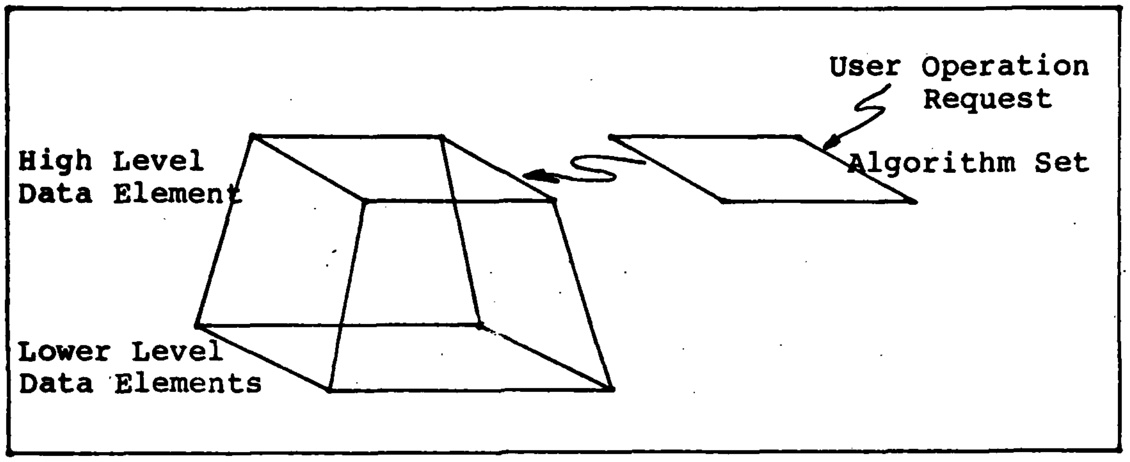


Fig 3. Data Elements and Algorithm Sets

algorithms which can be accessed via a single standard "call" can be considered by a user to be a single algorithm set. A data base manipulation program can do multiple functions, but is activated by a single call, and therefore can be considered by a user to be a single algorithm set. Each algorithm set is associated with a specific data element structure. Each algorithm implements all the desired operations on that structure, and is organized as in Figure 3.

"Algorithm sets" will be used in reference to those algorithms which operate on data elements. "Associativity" will be used in reference to data sets and data items, but must be defined first.

<u>Data Associations</u>. An <u>association</u> is any way that data items or data sets are related to one another. These can fall into two main categories, metric associations or non-metric associations.

A metric association is an association which can be mapped into a real number line, and has the property that differences between data item association values, or the association distances, are important to the problem being solved. A non-metric association is an association where the data association values can be mapped into numbers, but the resulting distances are irrelevant. The above two definitions do not cover all possibilities, but the subsets of computer data are assumed to fit these definitions.

Associations among data items must be representable in a convenient form to be easily used for further derivation. To represent associations, the terms "association value" and "association vector" are defined with a convenient notation.

An association value, V, is the numerical value into which an association is mapped for representation. An association description, S, is a representation of what the association means to the user. A data item, D_0, can then be considered to be associated with another data item D_1, and the associative link can be represented by (D_0, S, V, D_1). D_1 is considered associated to D_0.

A collection of data items, D_i, for i=1, ..., N can each be associated to D_0 via the same association description S. This will be represented by (D_0, S, V_i, D_i) where D_i associated to D_0 via S has the value v_i, i=1, ..., N.

A pair of data items can have many significant associations, and the number of associations, K, will be assumed to be finite. Then, by assigning an arbitrary order to the association descriptions, the association description vector, $\tilde{S}=(S_1, S_2, \ldots, S_K)$ will be formed. Also, the association vector, $\tilde{V}=(V_1, V_2, \ldots, V_K)$ can be formed. The association vector represents the association values of the K associations of the data item.

If D_0 is associated with D_i via each of the association descriptions, S_j, $j=1, \ldots, K$, then the full set of associations can be designated by $(D_0, \tilde{S}, \tilde{V}_i, D_i)$, $i=1, \ldots, N$. If all the data items D_i, $i=1, \ldots, N$, are associated to D_0 via $\tilde{S}$ then an associative data element can be described by

$$(\tilde{V}_i, D_i),\ i=1, \ldots, N$$

and D_0 becomes the reference for the data element, and the S_j are the descriptions for the K associations. D_0 and S_j, $j=1, \ldots, K$, become irrelevant to the storage and retrieval of the data items D_i once the association values, V_{ij}, are derived. Accessing the D_i is now a problem of finding the V_{ij} which are within a search region defined in a K dimensional vector space.

Data Access. Commonly known as "content addressing", associative access means "access by value". It is the process achieved by any of the number of search algorithms as found in Knuth's volume, The Art of Computer Programming, Vol 3, Sorting and Searching (Ref 16). However, most efficient search algorithms are based upon either data which is accessible via a mapping function, or are based upon data which is ordered in a single linear sequence, and thus mappable as a single association. Data which is always accessed via the same combination of associations has a _one-dimensional association_. Data which is accessed by K different combinations of associations has a _K-dimensional association_.

Regardless of the number of dimensions, associative access is expected to return all the data items which satisfy the search criteria. This makes associative access a _set-oriented_ access method, as opposed to _single element-oriented_ sequential access methods. Serial and random access are examples of single element-oriented sequential access methods. _Sequential access_ is characterized by data items accessed one at a time. The access may be in a _serial order_ or a _random order_. Sequential processes access data sequentially.

Computers are generally sequential processors of storage items. Users of associative access must convert the accessed set of storage items to a sequential form for use by the processor. This is often done in hardware by accessing only one of the data items at a time. Attempts at parallel processing are usually restricted to a small number of system processors due to interconnection complications and costs (Ref 30). Hardware implementations are very expensive for large applications, are sequential processors with various storage unit sizes, and are not readily available (Ref 30).

Summary of Basic Concepts. To implement associative data access, the associations must be modelled by data elements, and an algorithm set must be created to operate on the data elements. The algorithm set must be able to perform the associative access as well as translate an associative access data set into a sequential access data set for processing by a computer. Representing data association structures as mappings can make the algorithm set a set of map manipulation algorithms only. Thus, the associative algorithms and map are independent of all the characteristics of the data element which are not needed for representing the associations.

Mappings

Introduction. A mapping is a transformation from some input domain of values to some output range of values. Any input from within the domain results in an output within the range. Mappings can be pure algorithms, as in hash functions or numeric functions, or they can be predominantly data structures, as in virtual memory maps, tree structures, or vectored directories.

Mapping Algorithms. An N-M mapping is an operation which inputs N fundamental data elements ordered as an N-tuple, and outputs M fundamental data elements as an M-tuple. Each position, in the N-tuple and the M-tuple, is taken up by a specific type of fundamental data element which can only take on a specific domain or range of values.

The set of N-tuples which includes all possible combinations of values for the N fundamental data elements is the N-dimensional domain of the N-dimensional mapping. The result of the mapping will be an M-tuple of fundamental data elements, and each of the M fundamental data elements will have a set of values forming its range. The set of all possible M-tuples resulting from the mapping of all the points of the N-dimensional domain is the M-dimensional range of the N-M mapping. For Pascal record data, the mapping input may be an index into a table, and the output may be an address from the corresponding location in that table; once again mapping into the one dimensional memory address space of the computer. In the real world, an input could be a street address and a query as to location of the address, while the output is a string of instructions on how to get to that address in an automobile. A mapping can be a simple 1-1 mapping in a one dimensional space, or it can be far more complex.

Mapping algorithms in computers take on the characteristics of being memory-oriented, numeric, or structured. Some samples of data forms and their mapping requirements are given in Table II. Memory-oriented mapping algorithms "remember" their current location, so the user must navigate through the range space, always continuing from the last referenced location. Numeric mapping algorithms use a numeric function to generate numeric range

TABLE II

Data Forms and Applicable Mappings

Data Form	Computer Mapping Types Which May Be Used		
	Numeric	Structured	Memory
Array	Yes	No	No
Heap	Yes	Yes	Yes
Tree	No	Yes	Some Variations
Stack	No	Yes	Yes
Sequential file	No	No	Yes
Random access file	Yes	Yes	Yes

values from numeric domain values. <u>Structured</u> <u>mapping</u> <u>algorithms</u> use a <u>map</u>, which is a data structure such as a tree or a linked list, to map input criteria into the range space. Memory-oriented mappings may not require an input, and thus may be zero-dimensional mappings. The other mappings are N-dimensional, with N greater than or equal to one.

Mapping characteristics will be further discussed as the data element organization and access methods are discussed. The data element organization can be classed by how the data structure is accessed "easiest". The three organization access methods most commonly used are serial access, random access, and associative access.

<u>Data Element Structures</u>. To analyze data elements, the different structures can be categorized as though each data element were a map for finding lower level data elements, and a set of lower level data elements. An operation on a higher level data element can then be considered as a set of operations performing the following functions:

1. Retrieval of or access to the lower level data elements via the map,
2. Adding and removing lower level data elements, and
3. Manipulating the contents or values of lower level data elements.

The first two of these can be considered the alteration and use of the map, each using the map for finding specific lower level data elements. The third item is totally dependent on the lower level data elements and their associated algorithm sets. Using these operation requirements as differentiators the data elements can be categorized as simple, compound, or complex.

A <u>simple data structure</u> is a structure such that a single algorithm set exists which can perform every desired operation on that data structure, and is illustrated as in Figure 3. The definition is independent of the amount of information present in the structure. A simple data structure can be a single fundamental data element; such as a measurement, an index, a Boolean flag, or a character; or it can be any structure for which there is a satisfactory single algorithm set. Table I shows some examples of data elements which can be considered simple data structures. For real world data elements, the data structures can be far more complex than is realizeable in a computer, but still manipulated by a single algorithm set, such as a visual scene, analyzed by a single glance from a human eye. Simple data elements are the building blocks of all other data structures.

A <u>complex data structure</u> consists of a collection of data elements which cannot be handled by the repetitive application of a single algorithm. To create one common algorithm set, each lower level data element must be handled individually. Thus, at least two separate algorithm sets are required to process all the lower level data elements. Figure 4 shows the parallel hierarchies of algorithm sets and data elements for a complex data structure. The lower level data elements may or may not be of similar structures, and several examples of complex data elements are in Table I. The characteristic which makes it complex is that processing the lower level data elements requires multiple algorithm sets.

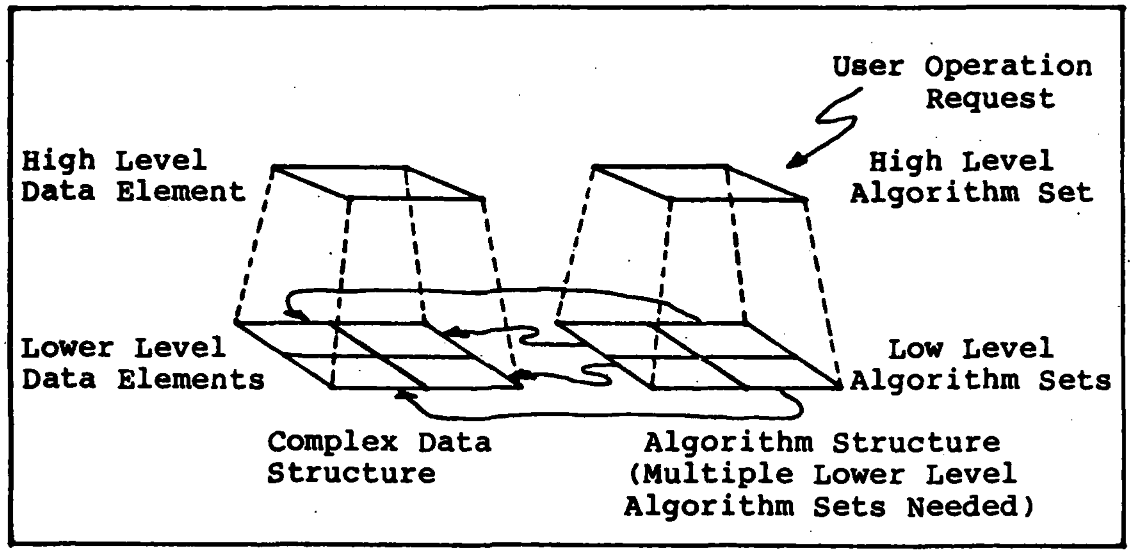


Fig 4. Parallel Complex Data and Algorithm Structures

A compound data structure is a collection of data elements which may each be manipulated by repeated applications of a single lower level algorithm set. Figure 5 illustrates the relationships between the hierarchies of the data element and the algorithm set. Table I gives several examples of compound data structures. The driving property here is that the compound data structure is constructed of a level of individual data elements which are treated algorithmically as though they are homogeneous.

Comparison: Compound Versus Complex Data Structures. Either compound or complex data structures can be used as simple data structures once the proper algorithm sets have been combined to create an algorithm set for the higher level data structure. The algorithm set for the high level data structure must include implementations of all the

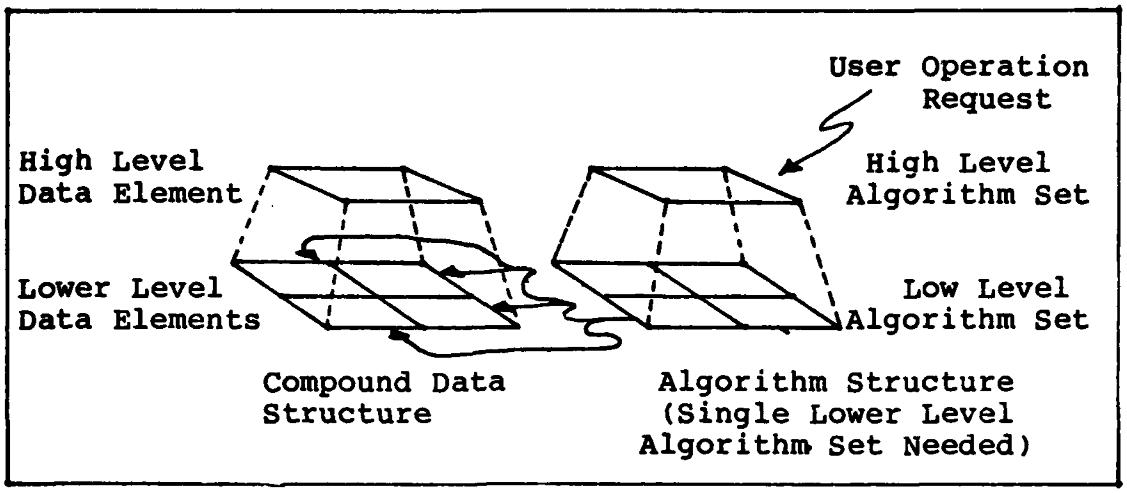


Fig 5. Parallel Compound Data and Algorithm Structures

needed operations on the data structure. These implementations include lower level data element removal and insertion, selection of the proper low level data element, and activation of an algorithm set to perform the necessary operations on each data element. To perform the needed operations on complex and compound data structures requires algorithms which can be grouped as follows:

1. High level algorithm set,
2. A selection or mapping algorithm for picking the low level data element on which to operate, and
3. The necessary low level algorithm sets.

Oppen (Ref 23) uses a similar approach to describe recursive data structures, but he uses only construction and selection algorithms in his definition. Here construction algorithms will be part of the <u>operation algorithm set</u>, and the selection algorithms will be referred to as mapping algorithms. The mapping algorithms may use a data structure of pointers, in which case the data structure will be referred to as the map. If selection involves more than mapping, then the remainder of the operation will be considered to be part of the operation algorithm set.

The characteristics of the high level user operations depend on the map that is used and the actions desired. The low level operations depend on the actions desired and the characteristics of the low level data elements. This leaves the mapping algorithms and the map, which will be discussed in the next section.

Associative Organization of Data

Introduction. To represent data associations in a computer, the associations must be modelled using the constructions available in the computer. In this section, associations among data elements are categorized and the methods of modelling the associations are discussed. Multiple associations between data elements are also discussed, and various resulting model structures are proposed and compared.

Data Associations. Data elements can be associated in many ways, and these associations can be categorized to aid in the simulation of the relationships in a computer. The main categories are logical associations versus physical associations.

A logical association is an association which depends on data content or some implied relationship between data elements. A real world example is the fact that plants and animals are both considered "alive", and thus a logical relationship exists between them.

A physical association is the occurrance of data elements "near" one another. In computers, using mapping algorithms, "near" can mean close in terms of some indices, even though the mapped actual memory locations are far removed from one another. A real world example of a physical relationship is a box and its corners, or a tree and its neighbor, one of each pair is physically related to the other. Working from logical and physical data associations further categories can be expressed and used for a better understanding of the data relationships.

Logical associations include measures of a single type which are close in value, but also include categories which are "close" or related ideologicaly in meaning or usage. Associations of "close" measurement values are simulated in a computer by values which are "close" in a numeric or alphabetical ordering sense. Thus content addressability is often used for simulating data associativity. However logical associativity also includes the relationships which are not easily quantized such as the word pair "feather" and "sneeze". These types of logical relationships must be modelled in a computer by physical links, since their relationship is entirely dependent on a logical model of some real event. Thus, through content associations by value and physical graph models, logical associations can be represented in computer memories.

Physical associations can be considered in two categories, association by proximity, for data elements which are "near" one another, and association by links, for those which are not necessarily "near" one another. The sides of a box are "near" one another, as are consecutive locations in computer memory, and thus are associated by proximity. Two cities may be connected by a single highway, and therefore associated by a link, much as a linked list in a computer memory. Both of these associations, proximity and link, can be considered the same if mapping functions are taken into consideration, because a link is a structured mapping. However, for the purposes of this paper, in computer data structures proximity will refer to adjacent memory locations, or members of a single block of memory, while link association will refer to association via address pointers, as in linked lists.

Both logical and physical data associations may be represented by combinations of proximity, link, and value associativity. These three forms of associativity are commonly used in computer algorithms and data structures, but they are not referenced as forms of associativity.

Serial access data structures use association by proximity. Association by value is used when the serial access data elements are ordered by value so that an efficient tree search can be used to retrieve specific data elements. This results in elements which are "close" in value being in physically "close" proximity to one another in the structure also.

Random access data structures allow the use of direct pointers to data element locations. This allows indexed access to any data element, and, through tables, trees, linked lists, or numerical calculations, allows many data elements to be "close" even though they are widely separated in the actual data structure, thus utilizing link associativity. Linked list and tree data structures (Ref 1; 10; 15; 34) are common examples of link associativity among data elements in computers utilizing random access. Tree structures and networks of links (Ref 10; 15) can give a multidimensional aspect to associations among data points.

Since computers are designed around random access memories, content associative access is a task of using actual proximity, value, and link association to retrieve data by simulating a pure content association from the random or serial access data. This has been attempted many times in many ways (Ref 17; 27; 30), and most ways included a number of exhaustive serial searches.

<u>Multidimension Data Associations</u>. Associations among data elements in a data structure may be categorized by the number of ways data in a specific structure can be related, or associated, to the other elements in the same structure. A serial array of data elements may be associated with one another only by proximity, or they may also be ordered and

thus also be related by value. The number of associations between data elements can be considered the dimensionality of the associative structure.

Association by proximity in a serial access data structure gives the data structure a one dimensional association. In random access data structures, such as arrays in memory, the association can be considered to be based on the number of dimensions of the array, ie. a four dimensional array would have a four dimensional association. In this way, the dimensionality of the association between data elements can be used as a measure of complexity of an associative data structure.

Multidimensional mappings can be used in random access data structures to map multiple indices into one dimensional memory address spaces. This allows multidimensional associations to be represented easily. The multidimensional mappings allow multidimensional proximity associations to be simulated through numerical mapping functions. These multidimensional mappings can also be simulated using tree or network structures of pointers which result in pointers into a one dimensional space. The pointer implementations of multidimensional proximity associations make the structured mappings more flexible than numerical mappings. Therefore, to simulate all types of associations, structured mappings are the most flexible tool available.

To use structured mappings for modelling associations, some association description conventions must be stated.

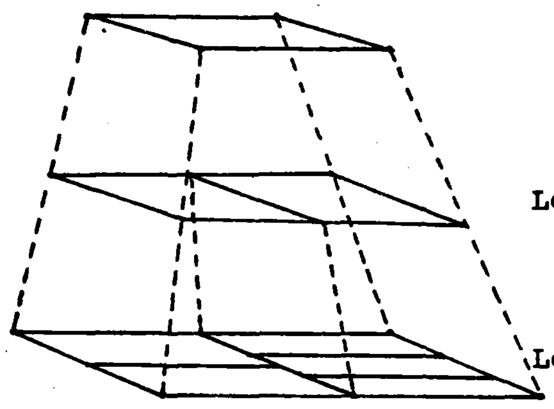
These conventions will describe the associative data sets in terms parallel to those of data elements, allowing easy modelling of data sets as data elements.

Data Set Forms. Groups of data, or data sets, having multidimensional associations may be organized in a number of ways. The three data base forms; network, hierarchy, and relational; are three varied approaches and will be used to illustrate the different forms of associative data. The main characteristic of interest here is whether the associations are homogeneous or nonhomogeneous.

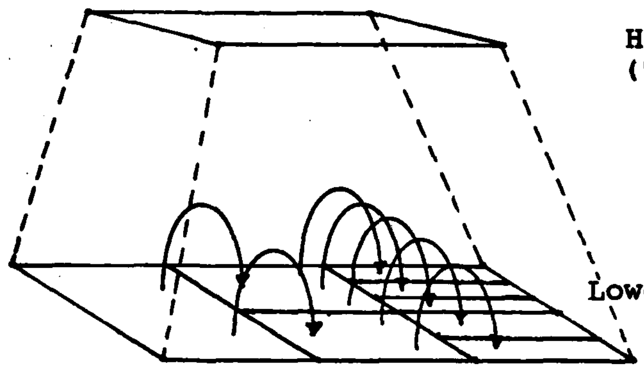
A homogeneous association of N dimensions is a compound data structure. There are N different associations, and every lower level data element is associated to every other by all N associations. In a relational database relation, this is illustrated by not allowing any null attributes in any primary key fields of a tuple.

A nonhomogeneous association of N dimensions is a complex data structure. There are N different associations, but at least one lower level data element does not use them all. More often, only a few of the N associations are used by all the lower level data elements. This type of association is prevalent in a network data base, where different data elements have different forms and thus are associated by links having different meanings.

A Hierarchical database, if observed in levels, can be considered a hierarchy of data elements as in Figure 6a,


High Level Data Element
Lower Level Data Elements (One Set, Homogeneous)
Lowest Level Data Elements (Two Sets, Each homogeneous)

a. Hierarchy of Compound Data Elements


High Level Data Element (The Whole Structure)
Lower Level Data Elements (One Set, Non-homogeneous)

b. One Level of Complex Data Elements

Fig 6. Two Views of Hierarchical Data Elements

with each level having homogeneous associations. However, an equally valid view of a hierarchical database is as a set of low level data elements, together making up a single high level data element as in Figure 6b, and thus has nonhomogeneous associations between the low level data elements. The point of observation, whether it is at a certain level within the data structure or at some point outside the data structure, determines whether the hierarchical database is a hierarchy of compound data structures, and homogeneous at every level, or a single level complex data structure, and nonhomogeneous.

Associative Data Models. Many simplifying assumptions must be applied before the volume of real world data can be reduced to forms which are meaningful and useful to model in a computer. Models of data sets with homogeneous associations are fairly straightforward, and this paper will limit itself to the analysis and implementation of such a model. Data sets with nonhomogeneous associations are not so easy, and will not be considered beyond this section. For more information on structures which are useful for modelling nonhomogeneous associations and manipulating those models see references on graphs (Ref 1:50-52; 10; 16).

Homogeneous associations can be modelled by low level data elements which contain N fields, one for each of the N associations between data elements. The data set then takes on the features of a relation in a relational database (Ref 9). The low level data elements become the tuples, and the

N fields representing the associations become the attributes for that relation.

For a one-dimension association, the model can be as simple as a one-dimension array of numeric or character values. For multiple dimensions, the model can be a file of of logical records or a table of records as in relational databases. In a sequential access data structure, the data may be ordered on the values in one of the fields or a combination of the fields, or it may be unordered. Random access data structures enable a more complex ordering using structured mappings to give a multidimension "nearness" to the data elements.

Data ordering becomes very important when the user starts evaluating time requirements for accessing lower level data elements. Ordered data can be accessed by binary search techniques in order(log(N)) time, while unordered data can only be accessed in order(N) time (Ref 16).

Associative Access to Data

Introduction.

Associative access to data has been seen as a panacea to mend all data retrieval problems, but without a fast response with correct and complete results, associative access is useless. A serial search and compare of all data in a data set is, technically, an associative retrieval. There are faster methods in use, and the proposed Associative Data Access Method will be compared to these.

The flexibility of an associative access method will be measured by how well it retains its favorable properties of speed and structure uniformity as the number of dimensions of associativity is increased.

One-dimensional Associative Access. Many applications take advantage of one-dimensional associativity. It is often used for accessing ordered sequential data in a random access mode. To use one-dimensional associativity is just to take advantage of the ordering and use a binary search, whether the search uses a numeric mapping, or a tree structured mapping, it is still one-dimensional associative access. The forms are common and can be found in Knuth (Ref 16). Their ease of implementation have given the one-dimensional associative access methods wide application.

The multiple binary trees in the relational databases force the user to access one primary key at a time. After selecting all the data meeting the search criteria on that primary key, the resulting retrieved data set has NO sorted or structured access available. For extremely large data sets, which result in large data sets after the first retrieval, any further selections on the data set will either be intolerably slow, or some structure must be generated to recreate the associative access previously available. Therefore, the _secondary selection process_ is the downfall of parallel one-dimensional access structures in multidimensional association applications!

To avoid losing the capability of further rapid associative access after a first retrieval, the results of a retrieval should be in an associative data structure. Also, to prevent having to perform numerous selections, one on each of numerous primary keys, A structure which merges the binary trees for searching on all primary keys at the same time would be advantageous. Such a structure would give rapid multidimensional associative data access.

<u>Multidimensional</u> <u>Associative</u> <u>Access</u>. Mapping each association into bounded numeric intervals, a multidimensional association then defines a region similar to a bounded vector space. Allowing the top level element in the associative structure to represent the entire region, then the next level down can split the region into equal parts. For K associations there will be 2^K equal parts, and each association will be split in half at that level. The result is a binary tree in K dimensions. To access the tree, the algorithm must move down to a level then perform a search of the smaller regions at that level. The search becomes one of comparing regions, the search region versus the small sub-regions defined by each node in the structure, as illustrated in Figure 7. For each node, the region it represents is either totally within the search region, totally outside the search region, or the two regions overlap. The search algorithm need only look further down the levels of the structure for those regions which partially overlap the search region. Those nodes which are fully within the search region are retrieved, intact,

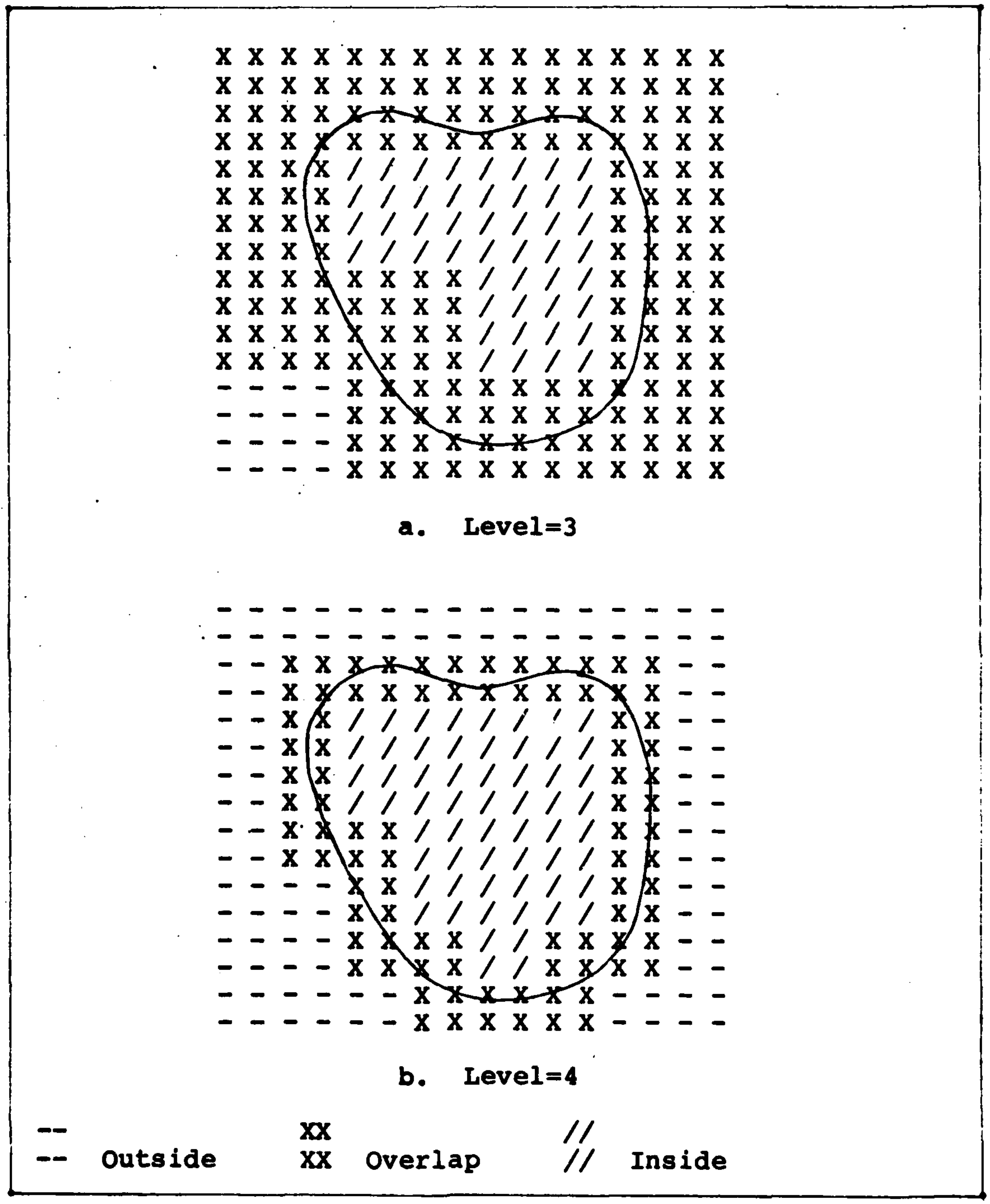


Fig 7. Node Regions Versus Search Regions, 2 Dimensions

into the retrieved structure. Those which are fully outside the search region are merely ignored.

The multidimensional associative access structure described has some interesting properties:

1. It can consist of uniform structure elements which make up the nodes of the associative tree.
2. At each level any one of a number of search techniques can be applied to select the proper sub-regions.
3. The structure consists of levels of homogeneous data elements.
4. The number of levels needed are determined by the amount of accuracy or number of significant binary decisions necessary to differentiate between the two closest data elements.
5. Only those nodes which represent regions containing data need be present at any level.
6. Data elements may differ in the value of a single primary key and still be represented as distinct from one another.
7. For certain applications, the difference between two numerically close data points may considered irrelevant if less than some set resolution threshhold. Thus, by setting a resolution threshhold, the number of levels can be limited and data points closer than the specified resolution will be mapped to the same point.

8. Searches for data can include multiple disjoint regions and still be performed in a single retrieval.

9. The entire associative access structure can be merely a mapping, allowing the data elements to reside elsewhere, including such media as paper files, magnetic tapes, books, or multiple floppy disks.

This versatile associative access method will be called Associative Data Access Method, or ADAM. ADAM will be analyzed, based on accepted performance criteria, then implemented.

Associative Data Access Method (ADAM)

Introduction. For associative access, ADAM will use a generalized set of associative access calls, defining the associative access algorithm requirements. The algorithms are designed to hide the actual implementation particulars from the user for ease of use. ADAM will also use a multidimensional associative map data structure to model the multidimensional associations.

Associative Algorithm Set. An algorithm set is needed to implement the necessary functions to be performed on the associative data set. This will be called the associative algorithm set. To perform associative access, the user must be able to CREATE the associative data set, ADD data items to the data set, FIND data items satisfying given search criteria, and to DELETE uneeded data items. To perform sequential processing on the items in an associative data set there must be a RETRIEVE function to convert a found associative data set into a sequential data set. The following is a formalized definition of these functions, using formalisms of Horowitz and Sahni (Ref 10).

```
structure  ADS(item,size,K,index)    Associative Data Set
           SDS(item)                 Sequential Data Set
           RDEF(index,value ranges)  Region Definition
    declare CREATE( ) --->  ads
           ADD(ads,item) --->  ads
           FIND(ads,index,rdef) ---> ads
           DELETE(ads,index) --->  ads
           RETRIEVE(ads,index) --->  sds
           EMPTY(ads) --->  boolean
```

for all m in ads; i,j in index; DIV in item; RD_i in rdef; s in sds let

```
           EMPTY(CREATE)::= true
           EMPTY(ADD(M,DIV))::= false
           DELETE(CREATE)::= CREATE
           DELETE(ADD(M,DIV),i)::=
                if DIV in $RD_i$ then DELETE(M,i)
                else ADD(DELETE(M,i),DIV)
           FIND(CREATE,i,$RD_i$)::= CREATE
           FIND(ADD(M,DIV),i,$RD_i$)::=
                if DIV in $RD_i$ then
```

```
                        ADD(FIND(M,i,$RD_i$),DIV)
                    else   FIND(M,i,$RD_i$)
                RETRIEVE(CREATE,i)::=  ∅
                RETRIEVE(ADD(M,DIV),i)::=
                    if DIV in $RD_i$ then
                        union({DIV}, RETRIEVE(M,j))
                    else  RETRIEVE(M,j)
    end
end
```

The functions EMPTY, CREATE, ADD, DELETE, FIND, and RETRIEVE are the operations a user should expect for any data set where data is to be stored, later retrieved, and possibly deleted. The FIND and RETRIEVE functions are often treated as a single function. Because RETRIEVE is a translation function from an associative data set to a sequential data set it is treated here as distinct from FIND.

This formalized definition not only gives a user's view of how the operations are used, but also define a set of functional tests. The functional tests will be the basis for final software acceptance, the prelude to performance measurement.

<u>Associative</u> <u>Map</u>. The ADAM associative map is a hierarchical data structure where each level is a compound data element. The overall map structure is made up of uniform data elements which form a set of homogeneous data elements, making the map structure a compound data

structure. Binding the manipulation algorithms for the homogeneous elements with the access operations defined above turns the associative map into a simple data element.

The map, with its algorithm set becomes an associative mapping from the K associations of the data items to the stored indices which uniquely represent the rest of each data item. The index may be the entire value, or it may be a pointer into a separate sequential data structure, allowing random access to its elements via the retrieved data set of pointers.

ADAM Summary. ADAM is an associative access mapping from K keys to a single data item. The data items should be accessed on any combination of keys in approximately equal time.

Associative access via ADAM will be restricted to data sets containing homogeneous data elements, each with K keys. It will be a K dimensional access mapping, and can be applied to databases requiring a fast, yet space-efficient, solution to the secondary key access problem.

Quality Assurance Requirements

To assure the highest quality software, good software engineering techniques will be used. The design is to be top-down, and the code is to be modular, with a high level of cohesion. All mainline functions of modules will be performed using passed variables. However, some debug and input/output operations may use COMMON or global variables.

The software package will include the ADAM algorithm set as its nucleus. However, to allow easy developement and debugging, and to create a user training program, an interactive application program package will be implemented. Each associative data access operation will be interactively callable, with complete error checking on all inputs. The user will also be allowed to dump the ADAM data structure to display or print, as well as turn on and off some function trace printouts.

Recursive procedure calls will be avoided. Also the hidden allocation of data storage will be avoided. This is to allow easy translation of all the program modules to low-order languages such as FORTRAN or assembly language.

The program package will be implemented in a manner which allows its use on small computer systems as well as large. For the purpose of this portability and ease of stating data descriptions Pascal will be the language used.

Summary

The ADAM hierarchical map structure and associated algorithms will be implemented in Pascal in a form easily translatable to lower order languages such as FORTRAN. It will be designed with portability, modularity, and maintainability as prime considerations.

An interactive debugging and training package will also be designed and implemented. This package will include "HELP" lists of legal commands and meaningful user prompts. The package will also test all user inputs for validity before attempting to alter the ADAM data structure.

III Design

Introduction

This chapter presents the overall structure and algorithm design of ADAM. Though the design started first, the implementation followed close behind. Many of the design decisions were chosen because of difficulties found in the implementation of the procedures in the Pascal language.

The evolutionary sequence of design decisions is presented, then the design itself is presented. In the development and design of ADAM a number of design tradeoffs were considered. The final product is one version of ADAM, but other versions are discussed.

Design Evolution and Tradeoffs

Introduction. An ADAM data set is a data element and its associated algorithm set. To implement it in a computer requires resolving the computer representable data structure to use, and determining the algorithm operational requirements. The design evolution presents successively more storage-efficient data structures for storing the data and allowing rapid access to collections of data points using data point "location" within the space, and the necessary algorithms to take advantage of the structures.

The evolution includes considerations of a "full space" data set with numerical mapping or hash function access, then hierarchical access. Hierarchical access is refined to multidimensional binary trees, and these are revised to allow unbalanced trees with missing data regions. Once the overall associative access structure is described, then ways to minimize the storage for each node are discussed.

Full Space Representation. N linearly ordered, equally spaced data points can be mapped to the N indices in [1,N] using

$$i = 1+(v-m)*s$$

$$s = (N-1)/(M-m)$$

where

i is the data item index,

v is the data item key value,

m is the smallest data item value,

s is the scale factor,

M is the largest data item value, and

N is the number of data items.

Multidimensional ordering of equally spaced data points can be stored in a multidimension array of locations, and accessed by numeric index mappings via a hash function. This is the most rapid way to associatively access fixed precision multidimension vectors (Ref 25). However it takes 2^{LK} data locations, where L is the bit length of each association dimension value, and K is the number of dimensions. A 3-dimensional region with 15 bits of accuracy requires $(32768)^3$ locations to store all the possible combinations as an array.

The ADAM is designed assuming that such an association space will be less than 1% filled! This assumption may preclude its use in some applications, but ADAM is intended to fill the gap, and be a practical, space-efficient way to represent and access data in such a space. Assuming less than 1% fill would leave an array representation with 99% wasted space.

Hierarchical Access. Binary tree access to data items commonly is height balanced around the data points present to minimize the number of levels required for a search (Ref 10:442). In ADAM the multidimensional binary tree data structure is region balanced around the normalized ideal association space. This ideal "box" in K dimensions is then subdivided into further ideal "boxes" at each successive lower level. For data of a fixed precision, the region balanced tree is restricted to the depth required to represent that precision. ADAM gives, rather than a minimum number of compares to access a particular point, a fixed number of levels of compares for any given precision data set. Height balancing a tree structure over data points requires adjusting the tree form as points are added or deleted. Region balancing, however, places the root node over the entire region, then divides the region up into equal parts, each part being represented by a region balanced subtree. Adding new data points and deleting old ones in a region balanced tree affects only the path from the root to the new or old data point. Thus, points can be added and deleted without affecting the whole tree. This structure stability allows region balanced trees to be accessed by multiple users, and thus, can allow dynamic associative access maps with multiuser properties. Region balancing also lends itself well to multidimensional binary decision processes.

Multidimensional Binary Trees. If, at each level, each dimension is divided in half, then each dimension undergoes a binary search in the process of searching from the top level to each successive lower level. At each of these levels, there are 2^K possible subtrees, each representing one of the possible combinations of decisions in K dimensions. There are several ways to access these 2^K possible nodes at a single level. Among these ways are the CARTAM structure, the K-D trees, and m-way trees.

The overall structure will be referred to as the major tree structure. Each level may have its own structure, depending on the method of representation. The structure at each level will be called the minor structure. The minor structure differentiates between the methods as follows:

1. CARTAM has a linear linked list as a minor structure,
2. K-D trees have a binary tree minor structure, and
3. M-way trees may have any minor structure.

CARTAM's minor structure is a linear linked list (Ref 24). Missing subtrees need not have a node in the linked list. The number of nodes required varies from 1 to 2^K, one for each subtree present. The search at a level is linear, requiring at most n compares for a retrieval when n subregions are present.

Associative access via K-D trees uses a binary search at all levels. Bentley (Ref 3) height balances his K-D trees, however the structure could be used for region balanced trees also. The K-D tree uses a K level binary tree for its minor structure, where the ith level corresponds to a decision in the ith dimension. This form requires between K and 2^K nodes for its structure at each level in each subtree, and for a search decision in only the Kth dimension the entire binary tree at each level must be traversed.

ADAM uses the CARTAM linear linked list minor structure. This gives minimum storage utilization for levels with only one subordinate subtree. This also simplifies the search algorithm for each level to a simple linear search, irregardless of which dimensions are used for the search criteria.

This form of the ADAM tree structure could be almost represented by the m-way tree of Horowitz and Sahni (Ref 10:496). ADAM, however, allows a node to have a single child, while m-way trees, including quadtrees (Ref 5; 11; 12) and B-trees (Ref 10:499), require at least 2 children for each node. The nodes of ADAM, rather than dividing the entire data set in half, add one bit of resolution to each of the K dimensions at each level for each of the K associations represented. ADAM does not have a strictly binary tree structure when viewed in term of the levels, and for K dimensions and L levels of precision, ADAM retrievals can require between L and $2^K L$ decisions for retrieval of a single data item.

Missing Data. Allowing for data points which are missing can cause much space to be wasted. As shown above, array representations are almost useless in many applications due to the wasted space allowed for missing data points. The m-way tree representations also have this problem, but it is not quite so obvious or severe. Using a variation on the m-way tree definition of Horowitz and Sahni, the ADAM node form may be shown as follows:

$$\text{A node, T, is of the form}$$
$$n, A_0, (K_1, A_1), (K_2, A_2), \ldots, (K_n, A_n)$$

where the A_i, for $0 \leq i \leq n$ are pointers to the subtrees of T and the K_i, for $1 \leq i \leq n$ are the key values.

To reduce the storage requirements of the nodes some of the fields can be implicitly stated. If all the fields are present and the node is a fixed size, then n can be eliminated. Since ADAM nodes represent ideal regions rather than stored points, A_0 is not needed to point to the stored data. The K_i's can be implicit if all the A_i pointers are present, and the relative positions of the pointers determine their key value, as can be the case in ADAM. The keys in ADAM are merely the numbers from 0 to 2^K-1 for ADAM nodes at all levels. However, for missing data points, the pointers are not needed. Often, only one or two of the pointers may be needed, and the rest of the 2^K pointers will be wasted space. For values of K greater than 2, this approach wastes significant storage space. ADAM's solution is to create a linked list of only those pointers which are needed at each level, each accompanied by its key. The final forms of a node and a level in ADAM are as follows:

- A node, T, is of the form

  (P,K,A)

- A level, L, is of the form

  $(P_1,K_1,A_1),(P_2,K_2,A_2), \ldots , (P_n,K_n,A_n)$

- The A_i pointers, for $0 \le i \le n \le 2^K-1$, point to the first node in a subordinate level or to data.
- The P_i pointers, for $0 \le i \le n \le 2^K-1$, point to the next node at the same level or to a parent node.
- for any level, $0 \le K_i < K_{i+1} \le 2^K-1$; $1 \le i \le n \le 2^K-1$ where K is the number of dimensions.

This form also allows the ADAM nodes to be of uniform size, eliminating many of the complexities of storage management for irregular blocks of storage, including complex algorithms, space needed for garbage collection, and possible storage fragmentation due to unuseable "leftovers". This same level structure of linked lists is used by CARTAM (Ref 24).

Retrieval Regions. CARTAM nodes, adapted for use by ADAM, would require 2 pointers per node, 1 region index per node, and 2K floating point numbers for the key in each node. The region index is added for specific ADAM applications in categorization. The two pointers are needed for suppression of unused pointers. This leaves the key representation as the only portion of the node which can be reduced in size using ADAM's constraints.

The 2K floating point values in the CARTAM key represent the subregion center value in each of K dimensions, and the region size in each of the K dimensions. The region size can be implicitly represented, and the center value can be shortened substantially. The techniques are presented next.

CARTAM restricts the total representable region to a finite range of values in each dimension. Also, each level represents a halving of the higher level. However, CARTAM allows levels to be suppressed if there are no branches. This allows terminal nodes to exist at higher levels throughout the structure. By not allowing the suppression of nonbranching levels, the size of the represented region can be implicit for the node level. For a node at level L, the size in each dimension of the subregion for that node is

$1/(2^L)$ of the entire region's size in that dimension.

The overall region size, since it is bounded and finite, can be normalized via a linear mapping to the half open range [0,1). Once normalized to this range, the Lth significant bit in the binary representation of each of the K numbers becomes the only needed key at the Kth level. The top level keys become the most significant bits of the K normalized values. Then, the next level's keys are the next most significant bits, and so on, to the bottom level, where the keys are composed of the least significant bits of the K normalized values. This bit-wise decomposition of the dimension values into level key bit patterns results in the numbers 0 to 2^K-1 as the possible key values at each level. This reduces the number of bits required for representing the region centers from K floating point words to K bits. For $K<16$ this results in less than one 16-bit integer word per key.

Map Storage. Often data storage includes not only the key data fields, but a number of data fields which, though not used for searches, contain information to be retrieved. In relational databases these non-search data fields are those attributes which are not considered keys (Ref 9:87-88). These will be called non-key fields. Date (Ref 9:87) defines a primary key as a field which is unique to every data item in the data set. For a large number of data items in any order in a file, on paper or disk, a primary key can be generated. The primary key can be as simple as the position index of the file in the set of files. ADAM is a mapping from the K keys to such a position index.

The ADAM data structure becomes a map from the K keys of the K associations, into the primary key position indices. To use the ADAM data structure, the user need only store the K keys and the value of the position index. The task of retrieving the entire data item then becomes two smaller tasks, that of associatively retrieving the data item index via the K association values, and the task of then retrieving the data item itself via the position index. Since the position index is a primary key, then the latter task is trivial, and may even be external to the computer. ADAM consists only of the associative mapping from the K key values to the position index.

Retrieval Forms. The sequential processing of a computer demands that the retrieval of the primary key indices be in a form sequentially accessible. However, sequential forms are not amenable to further associative access. Therefore, ADAM allows the user to FIND associative retrieval regions within its map as in Figure 8, and also allows the user to GET the indices of the retrieved data items in a sequential form as in Figure 9. Associative regions can be retrieved in either of two forms, parasitic maps or flagged nodes.

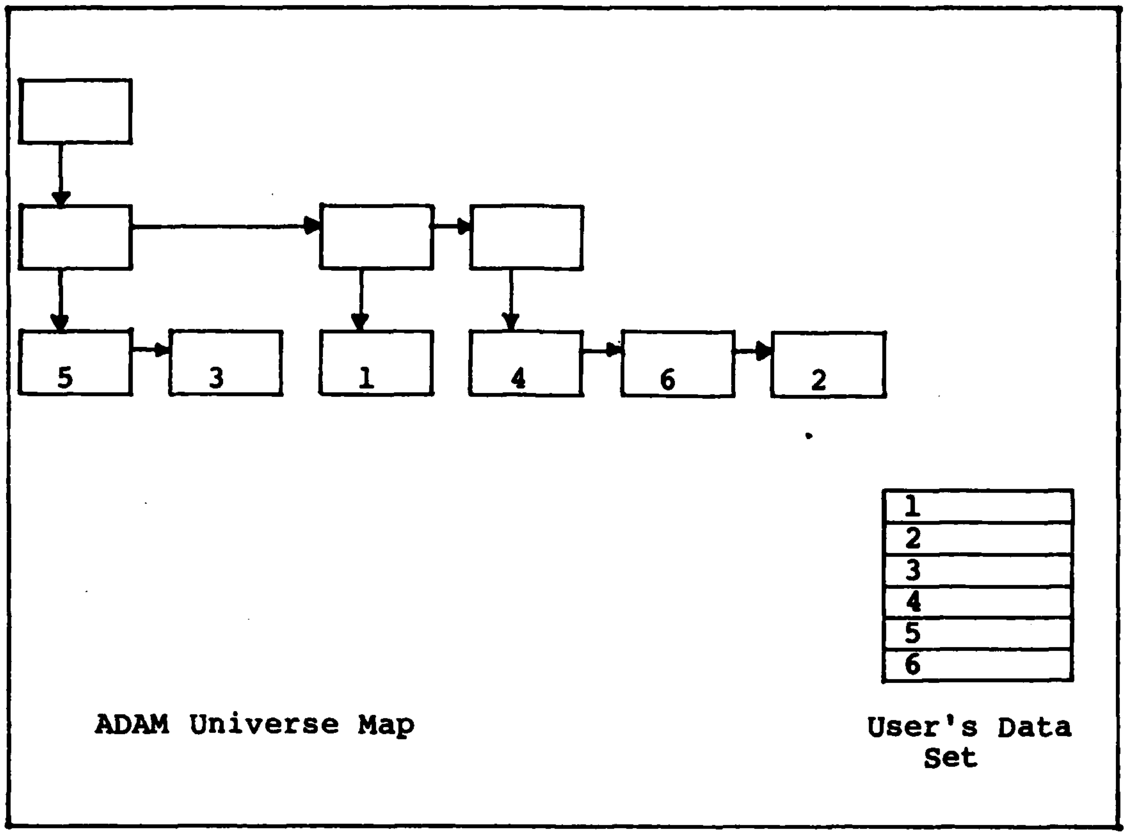


Fig 8. Associative Access via The ADAM Map

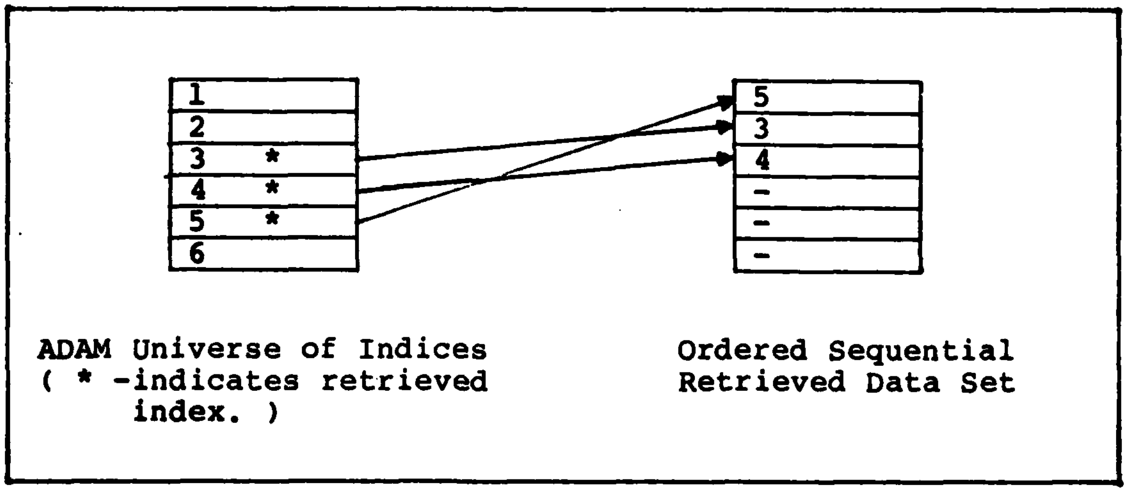

1
2
3 *
4 *
5 *
6
5
3
4
-
-
-
ADAM Universe of Indices (* -indicates retrieved index.)
Ordered Sequential Retrieved Data Set

Fig 9. ADAM Ordered Sequential Retrieval

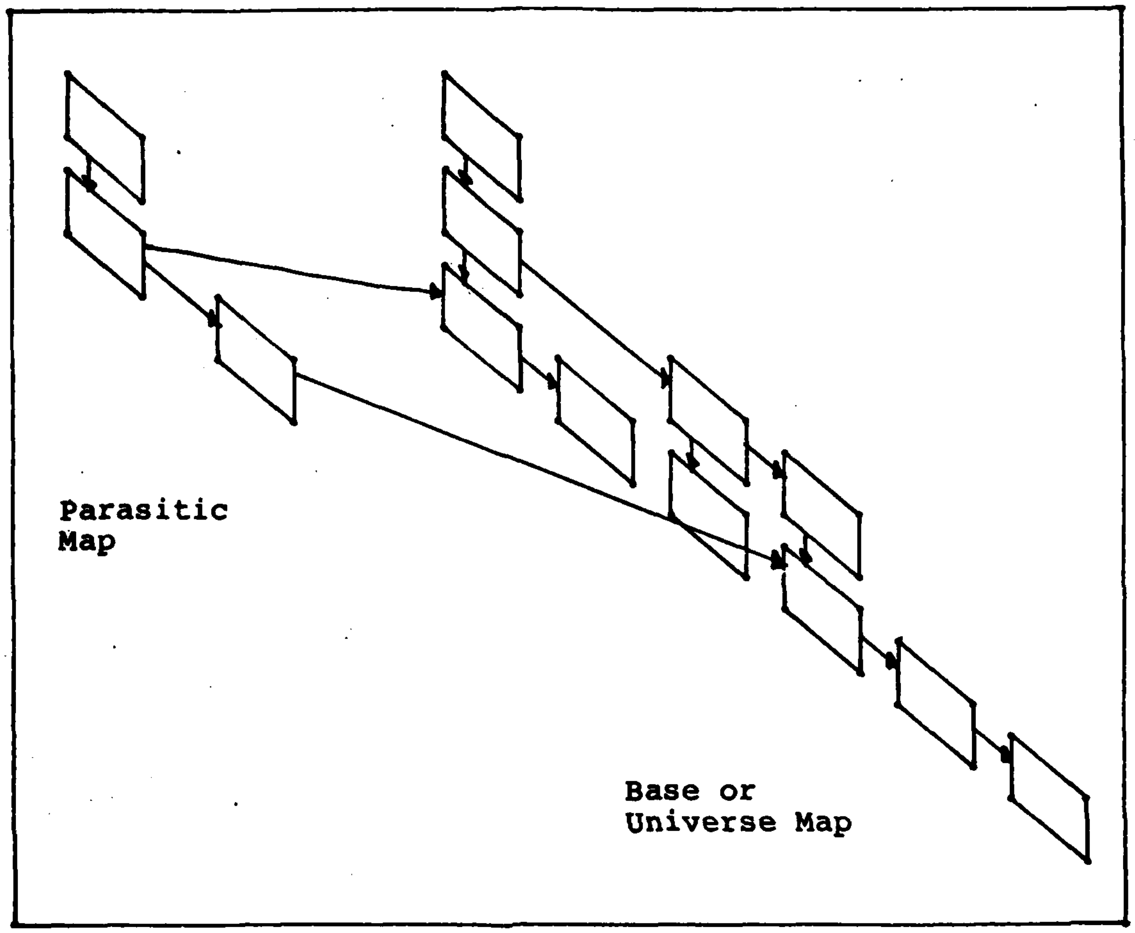


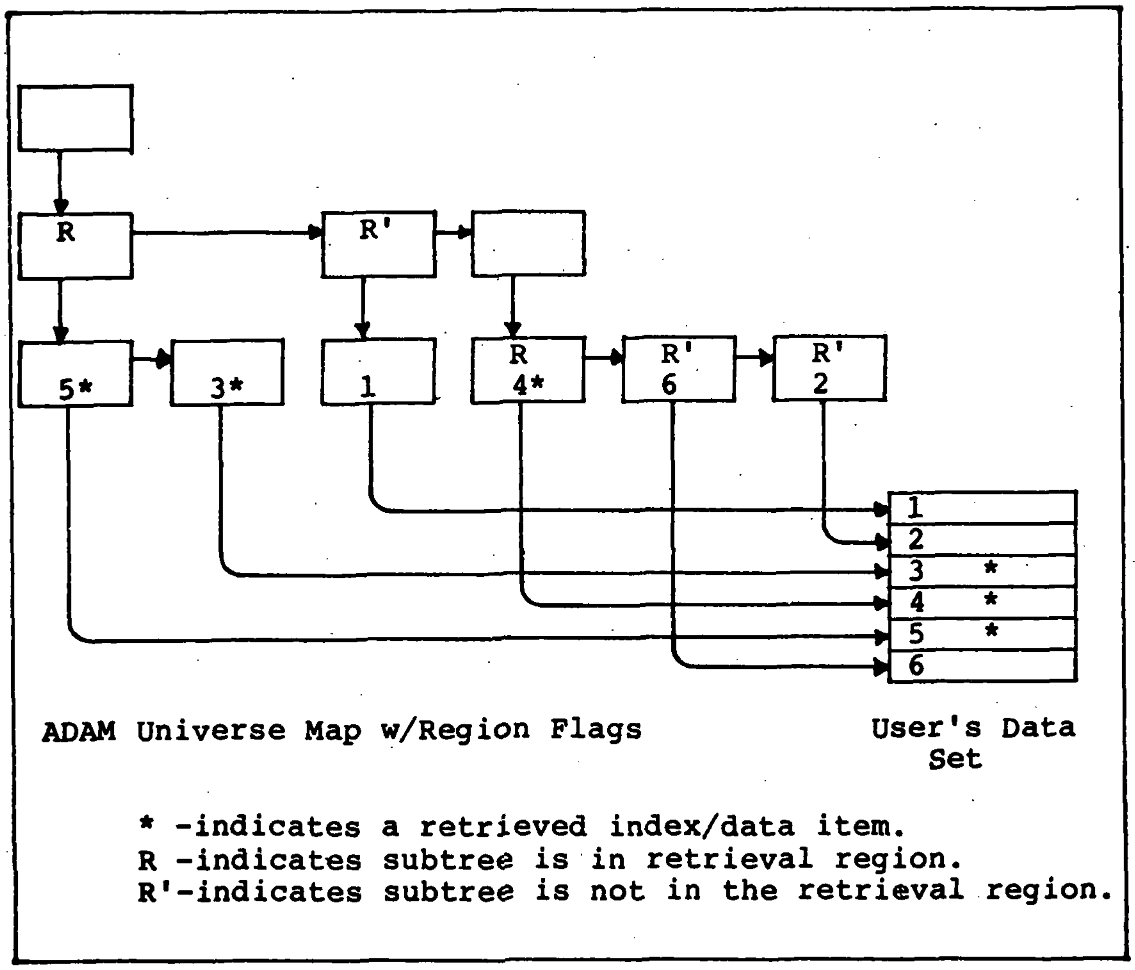
Fig 10. ADAM Parasitic Retrieval Map

A parasitic map is an associative map which has no terminal data nodes of its own. Instead, it has terminal references to nodes within a base map. The base map is the entire retrieval map, while parasitic maps point to subtrees within the base map. By pointing to only subtrees within their retrieval region, parasitic maps need duplicate only the base map structures which contain subtrees both inside and outside the retrieval region. Subtrees entirely inside the retrieval region are pointed to in the base map, and subtrees entirely outside the retrieval region are ignored.

Figure 10 illustrates a parasitic map with its references to the base map. Parasitic maps have the following advantages:

- They do not have to reside in the same map node buffer as the base map.
- They may exist in a user's file or memory space while the base map may be in some other user's or the system's memory or file space.
- One base map may have any number of parasitic retrieval maps.
- Set operations of UNION and INTERSECTION can be performed on multiple parasitic maps of the same base map.

The other retrieval form is via flagged nodes in the base map. Any number of regions can be represented this way, however the regions must be encoded by some method to allow for naming, region overlap, and representing large numbers of regions in small data fields within the nodes. This method of region representation is shown in Figure 11. Some of the methods considered for encoding the regions are region index encoding and region flag fields.



R
R'
5*
3*
1
R
4*
R'
6
R'
2
1
2
3 *
4 *
5 *
6
ADAM Universe Map w/Region Flags
User's Data Set
* -indicates a retrieved index/data item.
R -indicates subtree is in retrieval region.
R'-indicates subtree is not in the retrieval region.

Fig 11. ADAM Flagged Node Retrieval Map

The <u>region flag field</u> method consists of reserving 2 flag bits for each region within each node. The bits in a region flag field are interpreted as follows:

- Bit 1 =1 indicates the node region contains part of the retrieval region.
- Bit 2 =1 indicates the node region contains part of the complement of the retrieval region.

For each node the retrieval region can then be represented as follows:

- Field=10; Entirely containing the node subregion
- Field=01; Entirely outside the node subregion
- Field=11; Overlapping the node subregion
- Field=00; Unknown relationship beween the retrieval region and the node subregion.

The major drawback is that to maintain a small node size, the number of bits used to represent retrieval regions must be severly limited.

The alternative is <u>region index encoding</u>, where each combination of regions used is stored into a table. The index into the table is stored in the node to represent the region intersection combination for that node. This can be effective only when the number of region combinations used is small.

ADAM uses the region flag field approach, and limits its number of retrieval regions to 8 regions, requiring a total of 16 bits per node to represent the region flags. The parasitic maps were considered and rejected only because of time constraints in program development.

<u>Summary of Tradeoff Results</u>. ADAM assumes that the association space is a bounded finite region, and that the data points fill less than 1% of the space. To take advantage of the excess space and still give rapid access to the points of the region, ADAM implements a multidimensional levelled binary search tree similar to that used in CARTAM.

ADAM, however, taking advantage of some of the properties of the association space and the hierarchical structure to reduce the key sizes, uses much smaller nodes than those used by CARTAM. The resulting nodes in ADAM can be as small as 4 fields, each consisting of an integer word or less.

ADAM

Introduction. The Associative Data Access Method (ADAM) is designed to reflect an associative data mapping technique. An ADAM data element will consist of an ADAM map structure and an associated algorithm set for performing the desired operations on the map structure. This section presents an interpretation of the model used in the structure, the structure of the data element, and an overview of the basic algorithm set needed to build and access the data element.

Model Interpretation. ADAM is an associative data element which can serve as the basis for associative applications. For a real world data element of K associations to be implemented using an ADAM data element, the K associations must be mapped into a K-tuple of numerical values, with each association represented by one of the entries in the K-tuple. ADAM uses a rational numerical value in the half open interval [0,1), excluding the upper limit, for each association. For K associations the resulting ADAM structure will be a K-dimensional unit "box" as in Figure 12 (K=1, 2, and 3).

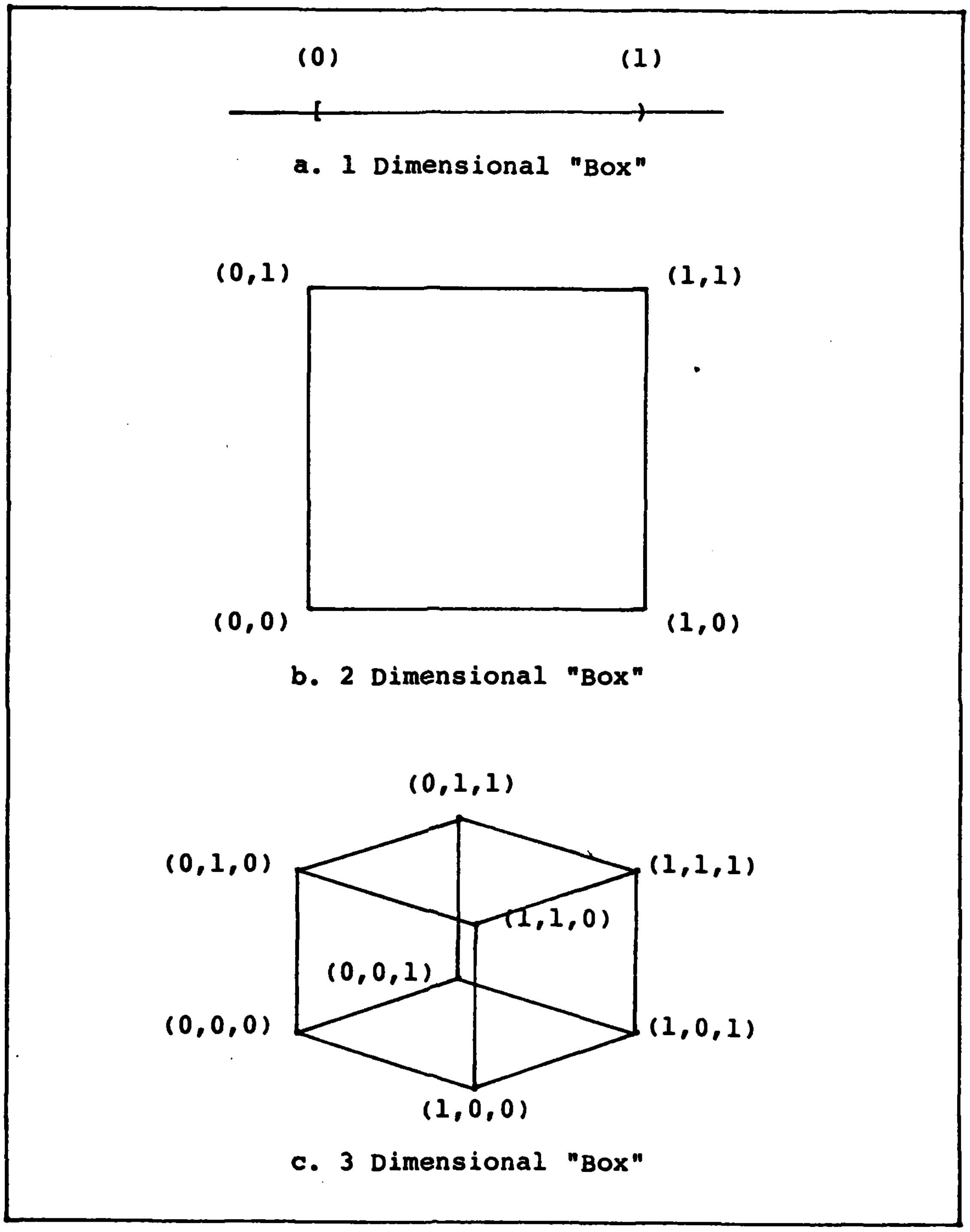


Fig 12. ADAM Unit "Box" Regions

The top level node of the ADAM map structure represents the entire unit "box". The lower levels represent progressively smaller subdivisions of the "box" into smaller "box" regions. Each lower level node represents a subdivision of the region represented by its parent node. Each dimension is "halved" at each lower level, such that at the top the intervals are [0,1) and at the first level the intervals are [0.0,0.5), and [0.5,1.0). The combinations of these intervals in each dimension give 2^K smaller "box" regions at each lower level for each higher level "box" region, or, for a 2-dimensional region, 4 smaller "box" regions for each higher region, as shown in Figure 13. Each lower level therefore, gives successively smaller regions, and thus, successively better resolution for locating points within the K-dimensional vector space defined by the top level region. A region at the 30th level measures $1/(2^{30})$ on a side. The 30th level region is within an overall space which is of length 1 on a side. This gives accuracy to about one in 1 billion, or 30 bits of accuracy.

Using the convention of "0" representing the choice of the lower half of an interval, and "1" representing the choice of the upper half of an interval, then the decisions at a single level, in all K dimensions, can be represented by a K-tuple of "1"s and "0"s. The region reached by that combination of decisions can be designated by the K-tuple which represents the combination of decisions. The K-tuple can be considered a K-dimensional <u>decision</u> <u>vector</u>,

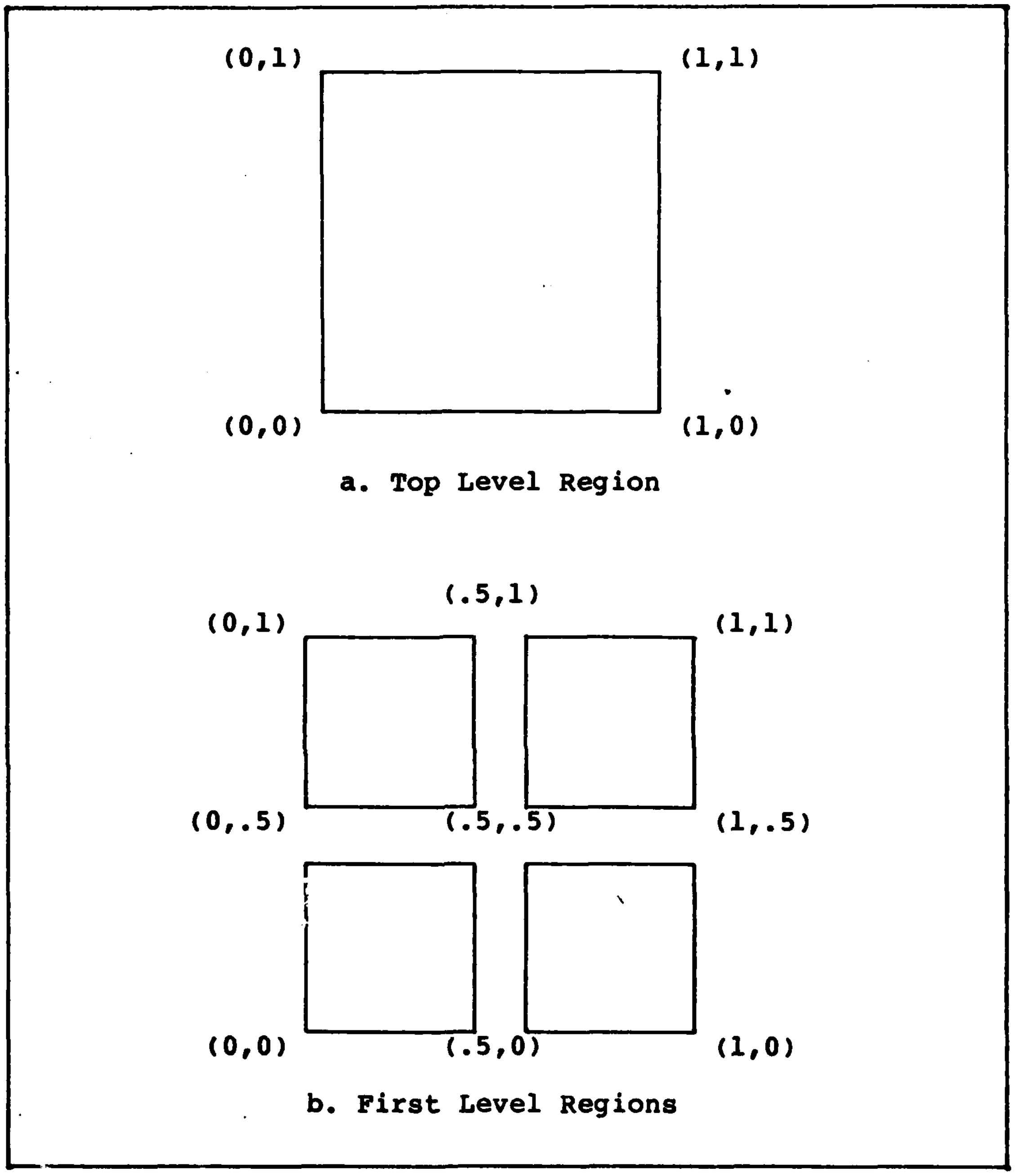
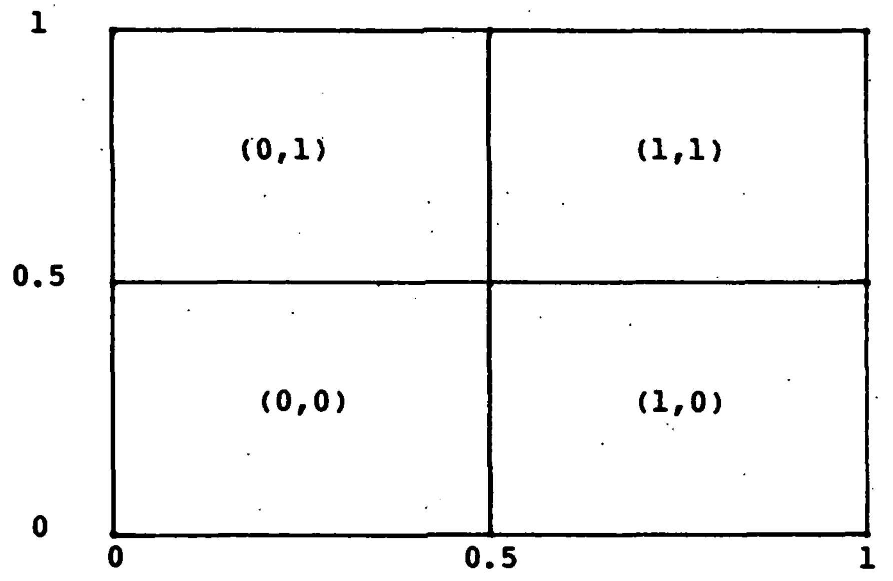


Fig 13. ADAM Region Halving by Levels

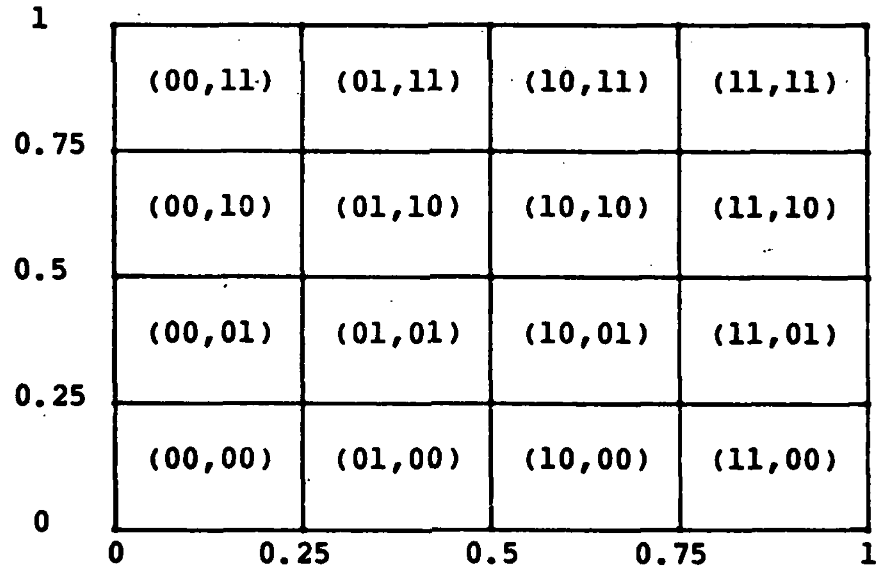
representing the decisions required to get to the region the association vector represents.

Figure 14a shows how the subregions of a single level of a 2-dimensional space can be represented using the decision vector notation. Figure 14b demonstrates the concept extended to three decisions in each dimension. By concatenating an additional bit for each decision onto the right end of the ith bit string in the K-tuple for the ith dimension, K-dimensional rational numerical vectors are formed. The vector components are bit strings representing rational numbers. The number of significant bits desired in the bit strings is the number of levels needed in the map. The bit patterns of the K bit strings, one for each dimension, represent the decisions required at each level to find the region in the ADAM map. These bit strings correspond to the normalized numerical value for the association, truncated to the number of bits for the level of the lowest decision.

The bit string vector is a decision "trace" for each dimension to any point, or data item, in the structure. The trace is the string of 0/1 decisions at each level to get to that point or data item. For 30 levels, the trace is a vector of K 30-bit numbers which represent the numeric location of the node within the normalized "box" representing the association space. This location vector is truncated at 30 bits, and thus gives 30-bit resolution



a. First Level Regions



b. Second Level Regions

Fig 14. Bit String Vectors for Region Designation

rational numbers. Points which differ only in the 31st bit position will not be discernable in a structure limited to 30 levels, and thus will be mapped to the same location in the structure. Arbitrary accuracy can be achieved by setting a specific limit on the number of levels in the structure.

This model displays the following properties:

- All data must be "normalized" to the half open numerical interval [0,1);
- The structure is hierarchical;
- The accuracy of the model is dependent on the number of levels represented;
- The access to associated data is a "top down" search process, starting at a root and searching the entire structure, as opposed to a "near-neighbor" process, which starts at any given position and searches the "nearby" points first;
- Each region node can be treated as a uniform data element, differing only in subregion size and location in the space;
- The physical interpretation of the structure can be a levelled K-dimensional binary tree, where each dimension is divided in half at each level.

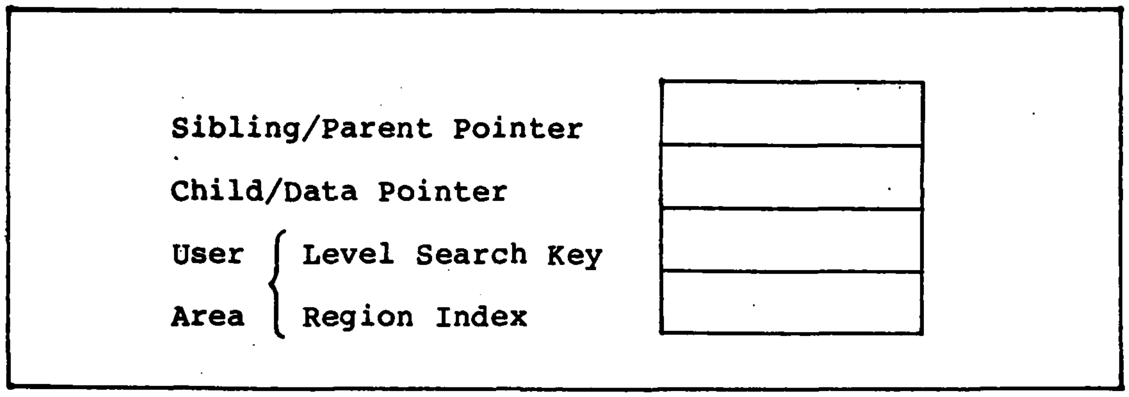


Fig 15. ADAM Node Structure

The Structure. ADAM uses a hierarchical structure. The structure consists of levels where any or all but one of the nodes at a level may be missing, yet all the nodes are uniform. Each node has two pointers, one to a twin node or parent node, and the other to a child node or data item. No unused or null pointers exist in the ADAM structure.

The nodes in an ADAM map are constructed as in Figure 15, and the user area contains the key field and the region flag field. The overall hierarchical structure can be viewed as levels of linked lists, with the last node at a level pointing back to the parent, allowing traversal either up or down the tree structure. The data structure is shown in Figure 16, and is similar to that implemented in CARTAM (Ref 24). Like CARTAM, if no data points are present in a region, then the node for that region need not be present in the structure.

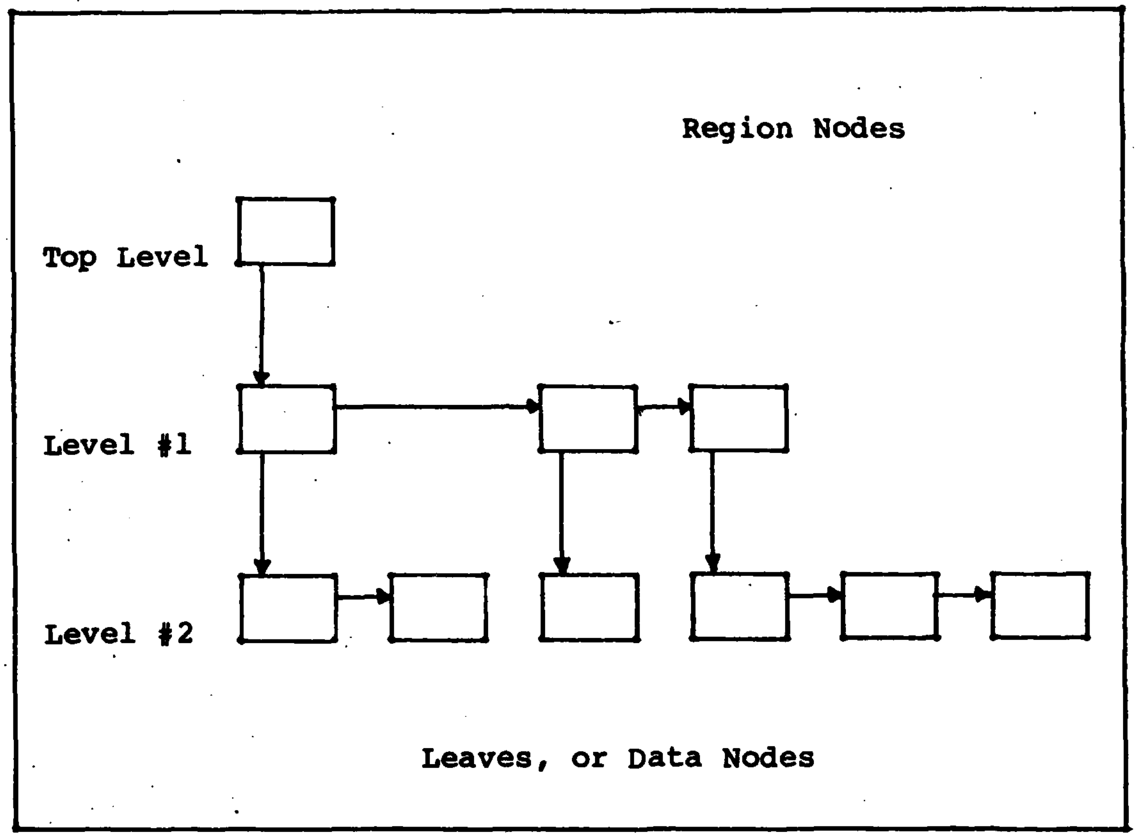


Fig 16. ADAM Map Hierarchical Structure

Each node represents a K-dimensional "box" region, and must have associated with it a region size and some indication of where it is within the association space. The size of a region is implicitly defined by the node's level in the ADAM map data structure. The position of the region in the association space is explicitely given by the "trace bits" in each node's key and in each key in the nodes at each higher level.

CARTAM stores both the center value and the "delta", or size, for each dimension, thus requiring 2K floating point numbers for each node. ADAM, by normalizing the data, can assume that a specific level always has a specific size region associated with it, and thus can rely on just a level index for the size of a node region. Also by normalizing, ADAM can assume that all data at level L under a single node will differ only in bit number L for each dimension, and thus the center value is determined by K bits. Using K bits in each node, representing the "search key" for the node, requires that every higher level node be present, but other nodes at the same level need not be present. Using the 10-level example from above, the two data items would require only 12 nodes to store them.

An alternative to the K-bit string is to store K traces in each node, however this substantially increases the node size over the K-bit string decision vector form. To suppress levels of nodes, the K-trace would have to be in each node, and a level index would have to be in each node. Small gains are made by each form for specific structure variations, but ADAM uses the K-bit string because of its small node size. The nodes will be of the form in Figure 15, with the K-bit string in the user field area.

The ADAM space is a K-dimensional vector space and includes all the possible values to be taken on by a K-dimensional vector. Each component of the vector is an association key value, and is a rational number representable by L bits and in the real interval [0,1).

These rational numbers are to be representable in the form

$$\sum_{i=1}^{L} b_i(1/2^i)$$

where b_i is an element of the set {0,1}. This gives the binary representation of the rational number as the concatenation of the b_i's, or

$$b_1 b_2 b_3 \cdots b_L$$

This, in turn, allows the trace vector to be expressed as
$(b_{11}b_{12} \cdots b_{1L}, b_{21}b_{22} \cdots b_{2L}, \cdots, b_{K1}b_{K2} \cdots b_{KL})$
and the level search key for the jth level is the jth bit of each of the dimension traces concatenated into one string, or

$$b_{1j}b_{2j}b_{3j} \cdots b_{Kj}$$

The Manipulation Algorithms. The ADAM map, to be useful, must allow the user to build the data access map, revise it, and selectively retrieve the data items from it. These operations are shown as ADD, DELETE, and RETRIEVE in the data flow diagram (DFD) of Figure 17. The DFD also shows the FIND operation, which is used to identify regions for later retrieval.

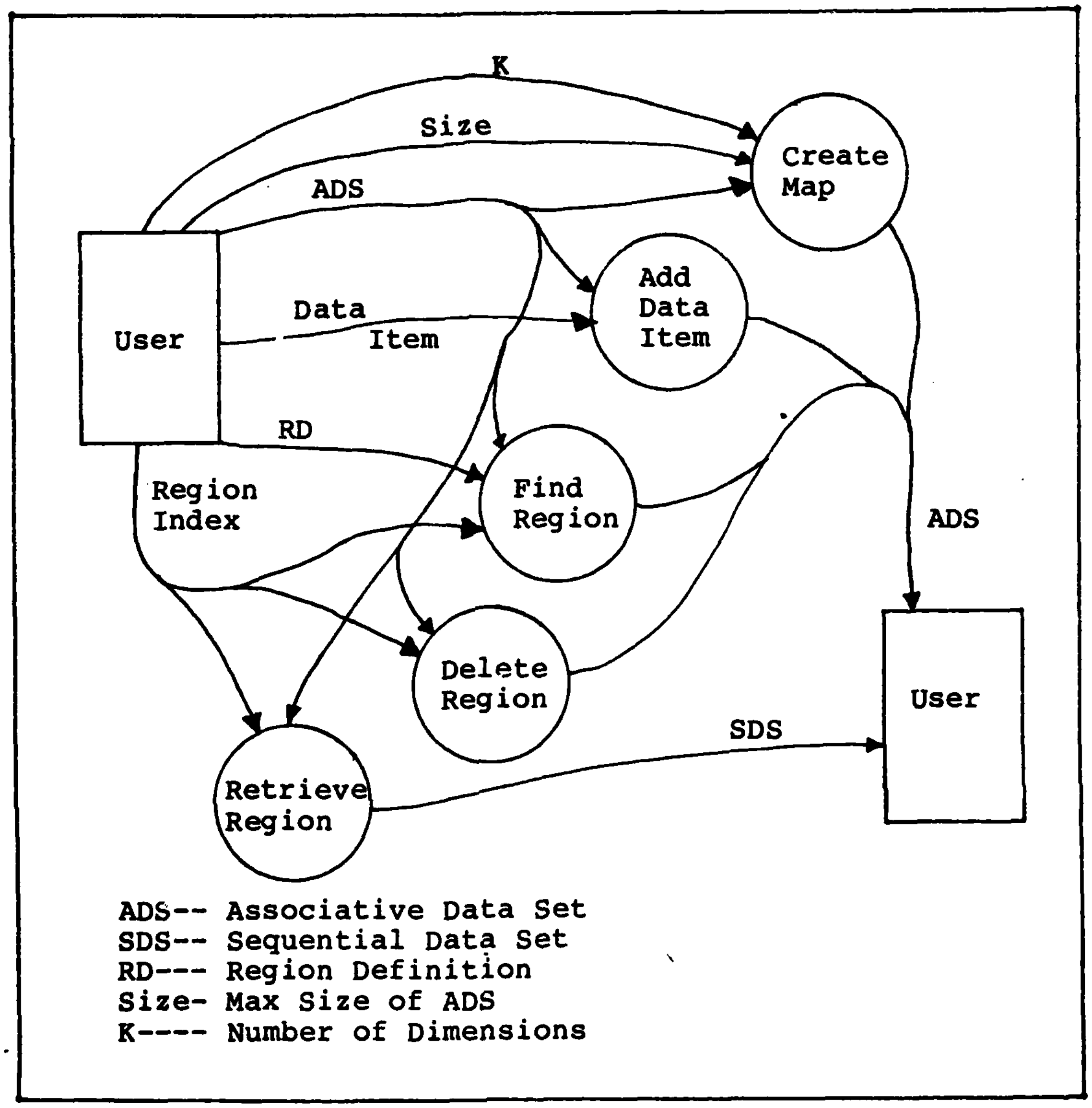


Fig 17. DFD of ADAM Algorithm Set

To ADD a data point to the ADAM map the K association key values, of L bits each, are converted to L K-bit level search keys. The L level search keys are then used to search for matches at successive levels. When a match is not found at a level, then a branch is built, storing the remaining keys in the lower level nodes. The new branch, if successfully built, is then inserted into the highest level where no match was found.

Both the DELETE and RETRIEVE operations rely on region indicators to determine which subregions to consider, but the region indicators are set by the FIND operation. To DELETE data points, first the region is selected by a FIND operation. All the data points within the region can then be DELETEd. The terminal nodes, or leaves, within the region are returned to free storage, and any higher level nodes which exclusively supported those leaves are also removed and placed back in free storage. To RETRIEVE data points, the region is also first selected using a FIND operation. The subtrees above and within the retrieval region are then traversed in their entirety. During the traversal the trace vectors are stored and the association keys are rebuilt. The retrieval can then return both the association key and the data item index for each data point retrieved.

The FIND operation depends on the region descriptions for its complexity. ADAM is implemented with only simple multidimensional rectangles. Complex data structures, data samples, or functions could be used to define retrieval regions. The FIND operation searches each level, comparing the subregions with the retrieval region. The comparison can give one of three results:

- The node region overlaps the retrieval region.
- The two regions are mutually exclusive.

- The node region is entirely within the retrieval region.

If the two regions overlap then the search must continue to lower levels. If the regions are mutually exclusive then the node and its associated region can be ignored. If the node is entirely within the retrieval region, then it can be flagged as such, and neither its nodes nor subregions need checked further.

The ADAM operations ADD, DELETE, FIND, and RETRIEVE as described fit the formal definition for an associative algorithm set are given in Chapter II, under the subsection on the associative algorithm set. ADAM customizes their implementation to handle the ADAM map structure.

Summary of ADAM. ADAM has some interesting properties. These can be grouped as structural, algorithmic, and application properties. The structural properties of ADAM are as follows:

- ADAM consists of uniform structure elements which make up the nodes of the associative tree.
- The number of levels needed are determined by the amount of accuracy, or number of significant binary decisions necessary to differentiate between the two closest data elements.
- Only those nodes which represent regions containing data need be present at any level.

The algorithmic properties are as follows:

- At each level a linear search is applied to select the proper sub-regions.
- Searches for data can include multiple disjoint regions and still be performed in a single retrieval.

The applications properties are as follows:

- Data elements may differ in only the value of a single primary key and still be represented as distinct from one another.
- For certain applications the difference between two numerically close data points may be considered irrelevant if less than some preset resolution threshold, and thus by setting a resolution threshold, the number of levels can be limited, and data points closer than the specified resolution will be mapped to the same point.
- The entire associative data access structure is merely a mapping, allowing the data elements to reside elsewhere, including such media as paper files, magnetic tapes, books, or multiple floppy disks.

ADAM is an associative access map, mapping the K keys of a K dimensional space to a single data item. ADAM allows retrievals as associative access maps for successive retrieval operations. It also allows retrievals as sequentially accessible data sets. Although retrievals could be as parasitic maps, ADAM has been limited to flagged node retrieval regions to reduce the program development time. Eight retrieval regions are allowed at one time.

The ADAM Program Package

Introduction. Three major methods of design were used to coordinate the program and data structures for ADAM. Data flow diagrams were used to coordinate the module interactions. A program control structure was used to show the hierarchy of control among the program modules. Also Warnier-Orr data diagrams were used. These allowed the specification of the data structures early in the design phase. These diagrams are given in Appendix C.

In line with the problem statement of Chapter I and the quality assurance requirements of Chapter II, the program was designed in modules using a top-down approach. Also, the program package was implemented for portability and use on small computers as specified in the requirements and problem statement.

Data Flow Diagrams. Appendix B contains the data flow diagrams for the ADAM program package. Data flow diagrams were used to coordinate the module/data interfaces and to organize the operation sequences. The lowest modules in the diagrams occur only as activities, with no further breakout for their sub-activities. The diagrams present the overall picture of data flows and transformations within the main modules of the system.

Program Control Structure. The ADAM program package is a collection of procedure modules, many of which are solely for the support of the five main procedure modules. To coordinate the module calls the hierarchy of control diagram of Figure 18 was created. The higher level modules call the lower level modules as indicated by the downward pointing arrows.

Data Structure Diagrams. The structure of each complex data element is described in the Warnier-Orr data structure diagrams in Appendix C. The data elements included are the following:

- User input data item vectors,
- User input region descriptions,
- User supplied map buffer area,
- Formatted ADAM map buffer, and
- Level search key/ trace bitstring structure.

These are not all the structures used, but they are the significant ones. All other structures use one of these patterns.

Overview. The ADAMTEST program package was written for machine independence, and to run on microcomputer systems as well as larger systems. The programs were written modularly for ease of implementation and later translation to other programming languages as needed.

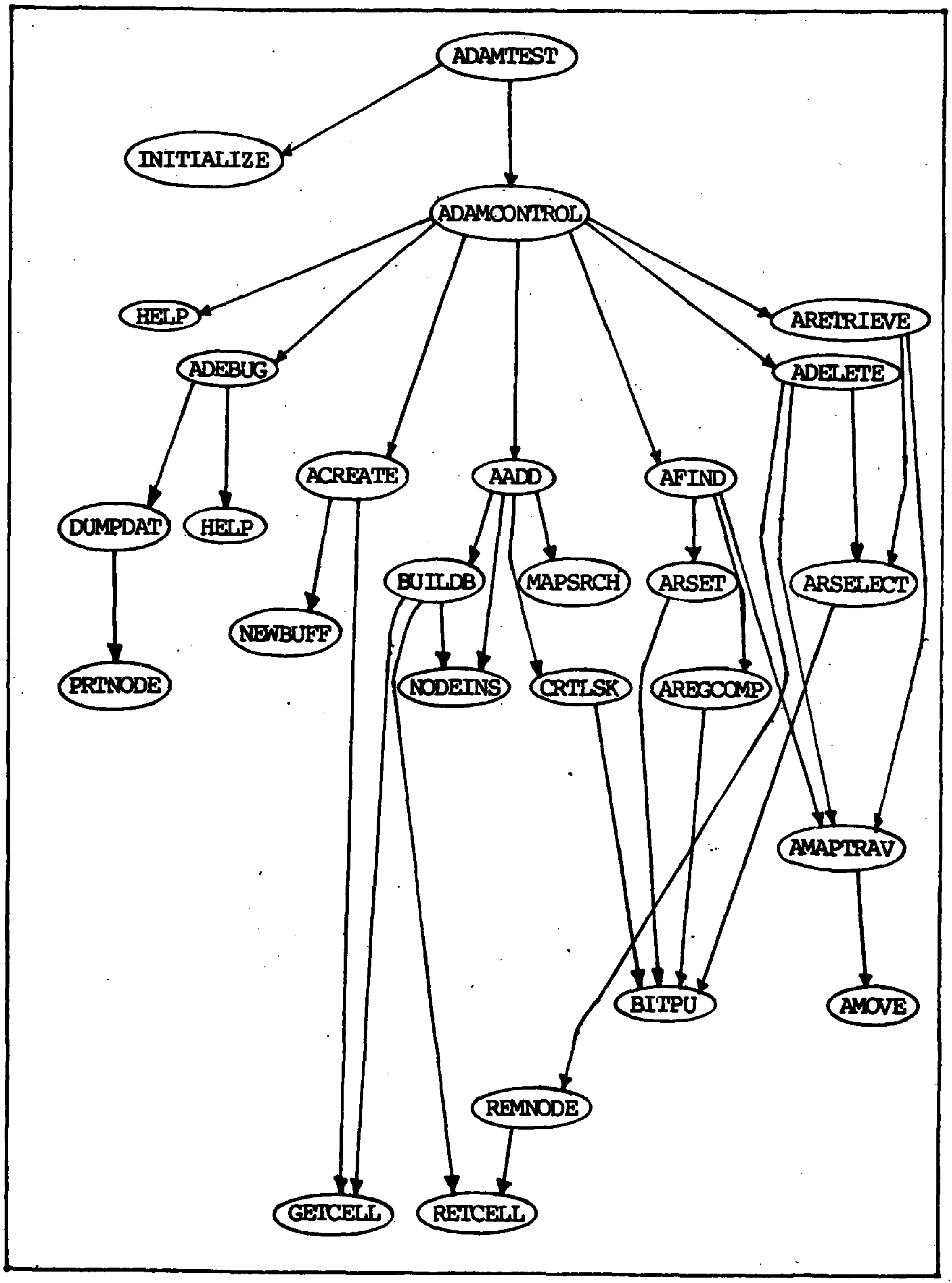


Fig 18. ADAMTEST Hierarchy of Control

The ADAMTEST package is implemented to run on a TRS-80 Model I, with 48K of user's memory, because student competition for AFIT systems makes their use inconvenient. The TRS-80 programs can be transferred as needed using an RS232 serial interface, or a telephone modem. The TRS-80 also made a good example of a common small computer system on which an ADAM program package may be useful. The programs were compiled and run using the ALCOR PASCAL compiler and runtime support package on the TRS-80, using three 5 1/4 inch floppy disk drives.

The ADAMTEST program package is modular, and includes an upper level of interactive procedures, allowing the user to activate the map operations in any desired order. A DEBUG module was included to allow dumping the map to the display or printer, and to activate certain trace functions in the different ADAM procedures.

The nucleus of the package consists of the ADAM map manipulation algorithms and their necessary support procedures. The package is written to run on any computer which has a PASCAL compiler following the language definition of Jensen and Wirth in PASCAL User's Manual and Report (Ref 14). Recursive procedure calls were avoided to allow easy translation of the ADAM procedures to other computer languages for general application.

Summary

ADAM allows the user to access data via a map. The map takes K dimensions of associations, or key fields, to a single index. This K dimensional mapping, when applied to metric data, simulates associative access by using a region balanced multidimensional binary tree. The major tree structure of the associative mapping is an m-way tree. Each node of the m-way tree has an internal structure called the minor structure. For ADAM the minor structure is a linear linked list, requiring a linear search at each level of the major tree structure. Data is retrieved as subtree structures, and a retrieval subtree can be translated to a sequentially accessible data structure.

The ADAM map requires the user to normalize all data to the half open interval [0,1) before storing it. The operations available are ADD, DELETE, FIND, and RETRIEVE. FIND returns retrieval maps, and RETRIEVE translates the retrieval map into a sequential form.

The map form gives the user a single index for each data item in the retrieval set. The actual data items do not need to be in the computer for ADAM to speed access to them. Indexed paper files, storage bins, or computer disk file records could be used as the final data storage to be accessed via the ADAM map.

Parasitic maps can enhance the usefulness of ADAM maps in a multiuser environment. The overhead storage for each parasitic map can be totally within the user's address space and disk file space, making it easier to keep accounts of user space and time requirements, and their usage of the data set. The current implementation, due to development time constraints, does not implement parasitic maps, but instead uses flagged nodes to mark retrieval regions.

The ADAM program package is implemented within an interactive control package called ADAMTEST. The ADAMTEST package allows the user to interactively activate each of the ADAM operations, and, using a DEBUG module, to inspect the map directory, map contents, and certain traced variables during the ADAM operations.

The five basic ADAM procedures are designed to allow an associative data type. The procedures perform all the necessary operations to allow the user to utilize the data type. The ADAM library allows the user to access many associative maps. Each map contains, in its directory, the needed information to manipulate the map, such as number of dimensions, number of levels, and size of storage buffer.

IV Implementation

Introduction

This chapter discusses the implementation of the ADAM procedures. The language and machine selected are justified, along with reasoning for algorithm portability. Finally the use of modularity is discussed, including module interaction, structure, and internal code characteristics. The organization of the program package is presented, including comments on the application of the ADAM algorithm set. The use of the interactive program package ADAMTEST is also discussed, as well as the use of the ADAM routine package alone. Good software engineering techniques are the driving force behind many of the implementation decisions. These were used throughout to satisfy the quality assurance requirements of Chapter II.

Implementation Particulars

Introduction. ADAMTEST was implemented on a single small computer, but intended to be generally useable on any large or small computer. It also was implemented in a single language, but intended to be translated to others easily by any programmer with moderate experience in the target language. Language constructs not easilly translated from Pascal to other languages were avoided. These decisions are software engineering techniques, and consider possible future uses of the program packag (Ref 26).

Environment. The ADAMTEST program and ADAM routine package were implemented in Pascal using a Model I TRS-80 with 48K of user memory. The compiler and linkeditor used was ALCOR Pascal, by ALCOR systems of Garland, Texas. The programs were run under the LDOS operating system, for the TRS-80, from Logical Systems Inc. of Mequon, Wisconsin.

Pascal was used because it uses strongly typed data structures and, by not using GOTO's, enforces rigidly structured code. Also, it is available on most large and small computers, using the Jensen and Wirth definition (Ref 14). FORTRAN was considered for its universality, but no structured version is available for the TRS-80, and it does not have the data structuring of Pascal. Other languages considered were ADA, not available; C, not universal enough; COBOL, not available; Z80 assembly, not universal enough; and BASIC, not structured or universal. Pascal also allows easy translation to and from structured English, and is thus easily translated to other languages.

Considerations. Once the algorithms are defined in Pascal, the conversion to other languages are easy. Certain capabilities of Pascal, however, are not easy to implement in other languages. These are recursive procedure calls, NEW and DISPOSE free storage management, and pointer variables. FORTRAN, COBOL, and BASIC have no universally defined recursive procedure calls. Free storage management is also not found in FORTRAN, COBOL, or BASIC. Both of the concepts as well as pointer variables, may be implemented at specific installations, but they are not part of the ANSI standard which is the universal subset of the languages implemented at most installations.

Recursive procedure calls were avoided by implementing the state machine in AMAPTRAV and by using an explicit position stack to move through the map structure. Position and data stacks were used in the procedures AADD, ADELETE, AFIND, ARETRIEVE, and AMAPTRAV.

Free storage management was the next complication. Pascal accesses storage as needed from the "left over" memory using the NEW procedure. In some implementations, Pascal also allows the storage to be freed again using the procedure DISPOSE. Languages such as COBOL and FORTRAN don't allow this management of undeclared storage in "left over" memory space. To insure complete control over the memory space used, the map buffer is explicitely declared in the calling routine, then the free storage management is initialized by a call to ACREATE. Having explicitely declared map storage also allows storing the entire map to disk by serially indexing through the buffer locations, allowing easy saves and restores of intact associative data maps.

Using a declared array of nodes for free storage, permits a node location to be referenced by an integer index. This eliminates the requirement for pointer variables which point to absolute memory locations in undeclared memory space. Languages such as FORTRAN, COBOL, and BASIC can implement the array storage and index integers, thus the buffer and index convention is better for translation to these languages.

Restrictions. The ADAM program library is restricted by the ADAMTEST application in both size and use. Through ADAMTEST the user can only interact with ADAM via the standard input, standard output, and the PRFILE list output. The size limitations make the ADAMTEST program package too restricted to apply. It is intended only for manipulating the ADAM map and debugging the ADAM routines.

The ADAMTEST package was implemented to allow interactive use for debugging the ADAM routine package, a software engineering technique for simplifying the testing process. The map buffer was restricted, for the ADAMTEST application, to MAXNODES=50 map nodes, which allows about 10 data items for a 6 level map structure. For widespread application the constant MAXNODES must be set to a value large enough to contain the desired data items. See Chapter V, Space Requirements, for an estimation procedure for ADAM map space requirement.

The number of bits is limited to 16 as the constant MAXBIT, and the number of regions to 8 as the constant MAXREG. These are set for the maximum number of bits that can be contained in a 16 bit integer, and can be adjusted as the integer size changes on other machines.

MAXDIM is fixed at 10. This gives a maximum linear search of up to 2^{10} =1024 compares at each level. Larger values may be unreasonable, and are discussed further in

Chapter V.

Program Organization

Introduction. The ADAM implementation is three major functional packages under the control of the ADAMTOP, as shown in Figure 18 of Chapter III. These functional packages are the ADAM routine package, the interactive user interface package, and the debugger user interface package. With minor modifications to remove debugger functions, the ADAM routines can be used without the others to support user applications.

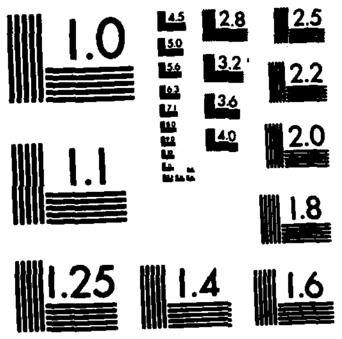
The ADAM Routine Package. The ADAM routine package, implements the associative algorithm set. The routine package is in four libraries, ALIB1, ALIB2, ALIB3, and AUTIL1. These libraries are listed seperately in Appendix D.

ALIB1 contains the ACREATE and AADD procedures for creating an empty ADAM map and adding data items to the map. AADD contains the procedure BUILDB, which builds a single *tree branch*, a subtree with only one path. BUILDB is only used by AADD. ACREATE and AADD also refrence procedures in AUTIL1 to perform operations common to ALIB2 and ALIB3. They also reference the variables and procedures which are used for debugging only.

ALIB2 contains the AFIND, ADELETE, and ARETRIEVE procedures for finding a defined region, deleting a found region, or retrieving a found region to a sequential retrieval form. REMNODE, a procedure only referenced by

AD-A124 674 ASSOCIATIVE DATA ACCESS METHOD (ADAM)(U) AIR FORCE INST 2/3
OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING
J R HOLTEN DEC 82 AFIT/GCS/EE/82D-19

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

ADELETE, is within ADELETE, while AREGCOMP and ARSET are only used and therefore are defined within AFIND. The procedures in ALIB2 reference the procedures in ALIB3 and AUTIL1 to perform common operations. This library also has references to debugger variables and interactive output routines.

The procedures in ALIB3 are ARSELECT, which selects the proper region bits from the region flag field in a node, and AMAPTRAV which traverses an ADAM map and returns status flags which indicate the type of node currently being visited. Another procedure, AMOVE, is in ALIB3, but it is only used by AMAPTRAV. AMOVE performs the actual position index adjustments necessary to move through an ADAM map. ARSELECT references the common routine BITPU in AUTIL1. ALIB3 does not contain any references to the debug or output routines, but does reference the global print file PRFILE.

The library AUTIL1 implements a number of the common operations required by the other modules in the ADAM program package. The procedures in AUTIL1 are grouped by operations which are performed as follows:

-Free node manipulation; GETCELL, NEWBUFF, and RETCELL;

-Bit manipulation; BITPU and CRTLSK; and

-Map manipulation utilities; NODEINS and MAPSRCH.

Only CRTLSK calls BITPU and references the debugger variables and output procedures, otherwise all of these procedures are "bottom level", calling no other procedures.

The ADAM routine package can be used as a stand alone library to be linked to a user application program, but for this project it was manipulated by the interactive control routines in the interactive user interface package.

The Interactive User Interface Package. To create, build, and use an ADAM map with enough flexibility to quickly debug the ADAM procedures, a high level set of control routines were used. ADAMTOP is the top level library of the entire program package. Its structure is shown in Figure 18 of Chapter III. The program name is ADAMTEST, and it contains the high level procedures ADAMCONTROL, INITIALIZE, and HELP.

INITIALIZE is called once to initialize global variables which contain needed constants. These are MASK, an array of bit masks; PRFLAG, the print/display routing flag; BITFLG, for enabling bit array trace printouts; and VECFLG, for enabling integer vector trace printouts. Once these variables are initialized ADAMTEST passes control to ADAMCONTROL which remains in execution until the command "STOP" is input in response to the "ADAM COMMAND=" prompt. The legal commands are in Appendix D under the ADAMTOP library listing of the HELP procedure.

The ADAMCONTROL procedure calls several procedures from the library AINTER1 to perform interactive input and output of specific types of data elements.

The only ADAMCONTROL command other than "STOP" which does not activate an ADAM map manipulation routine is "DEB". "DEB" activates the debugger user interface package.

Debugger User Interface Package. To interactively inspect the ADAM map after each operation and selectively trace certain variables during some operations, the debugger was implemented. Once activated, the debugger continues in control until "STOP" is given in response to the "DEBUG COMMAND=" prompt. The debugger resides in the library ADEBUG. The package consists of the main routine ADEBUG and its support procedures, PRTNODE, DUMPDAT, OUTDIR, and HELP.

The command "DIR" allows the user to dump the current map directory, and "DUMP" allows the user to dump the map buffer contents. The directory is dumped by a call to OUTDIR, while the map buffer is dumped by a call to DUMPDAT. DUMPDAT repeatedly calls PRTNODE to print each node in the interval of node indices the user selects.

The other commands of ADEBUG are given in Appendix D , in library ADEBUG, in the procedure HELP.

Summary of Program Organization. The procedures in the ADAM routine package are designed to stand independent of the interactive control package, but have had some debug functions inserted into them. To use them independently the debug functions must be removed.

Using the ADAM routine package, interactively or otherwise is the next consideration.

The Use of ADAM

Introduction. The ADAM routine package may be used interactively via the ADAMTEST program, with the full debugger capability, or it may be used seperately as a procedure library for supporting applications programs. To use it seperately, however, the debugger functions inserted in each of the ADAM routines must be removed.

Interactive Use of ADAMTEST. The ADAMTEST package is fully interactive. All user inputs have prompts, and there is a "HELP" command to get the list of commands printed out. For commands requiring additional information the user is prompted further. All inputs are checked for validity, although the checks are for range only on numeric inputs. Alphabetic inputs, when numerics are expected, can give results which may be implementation dependent.

Initially, printouts of retrieved data are routed to OUTPUT, or the display file. By activating the debugger using "DEB", giving the debugger command "LIST", and then "STOP", to return from the debugger, all further retrieval outputs will go to the print file PRFILE. The output can be returned to the display using the debugger command "DISP".

While in the debugger the trace commands "BITS" and "VECS" toggle trace flags, reversing their status each time the command is used. "BITS" activates the bit array trace printouts, while "VECS" activates the integer vector trace printouts.

"DIR" and "DUMP" debugger commands give the user a view of the current status of the map. "DUMP" prompts the user to input an interval of map buffer indices, allowing the user to inspect portions of the map buffer selectively, or all at one time. These both output their data to the current retrieval output device.

The user may wish to apply the ADAM routine package without the problems of interactive data entry. This is also possible.

Applications Use of the ADAM Routines. By linking LIB1, LIB2, LIB3, and AUTIL1 object libraries to a user package which has external references to ACREATE, AADD, AFIND, ADELETE, and ARETRIEVE the user can access the algorithm set directly.

The proper global data types must be declared in the user's calling procedures and passed in the ADAM procedure calls. These data types include ADAMMAP, PFILE, DIVEC, REGDEF, and SEQDS, as found in the declarations in the library ADAMTOP in Appendix D.

All references to the debug variables, interactive I/O routines, and global variables also must be removed from the ADAM routines. These include references to the variables PRFILE, PRFLAG, BITFLG, VECFLG, and MASK.

To speed up the ADAM routines, the procedure BITPU in the AUTIL1 library should be implemented in a language which allows rapid bit retrieval. Rewriting BITPU can eliminate the need for the bit masks in the array MASK.

A view of the module interactions is necessary to allow the user to modify the routines as needed for use with applications programs.

Program Modules

Introduction. A useful tool, the hierarchy of control diagram, Figure 18 in Chapter III, was created for showing the module control hierarchy. Using it, the interaction between modules can be graphically presented. The hierarchy of control for the entire ADAMTEST program is presented. The types of module structure are then discussed. Other useful characteristics of the module source code is then discussed.

Module Interaction. The hierarchy of control diagram shown in Figure 18 shows the pattern of procedure calls. ADAMTEST is the main program, calling INITIALIZE and ADAMCONTROL. From there control passes downward from those which have no further procedure calls.

This graphic representation illustrates that there are no recursive procedure calls. Programming languages such a FORTRAN and COBOL do not allow recursive procedure calls, thus, to ease translation into such languages, recursive procedure calls were avoided.

Although Figure 18 illustrates the hierarchy of procedure calls, this is not the only way the modules interact. Module coupling must be considered in more depth.

Almost all the modules have only "data" coupling, where all communications are through passed arguments. These exceptions are the print file, PRFILE; the debugger flags, BITFLG and VECFLG; the print flag PRFLAG; and the bit mask array MASK. These are all treated as "external" coupling, where the routines share a COMMON data item.

Module Structure. The good Software engineering techniques, required for quality assurance by Chapter II, dictated the size and cohesion of the modules (Ref 26). The body of each of the modules was kept below 100 lines of code, and thus is two pages or less in the source listing. Being short modules, the cohesion can be easily observed. In most cases the module cohesion is functional, however the exceptions are AMAPTRAV and AMOVE.

AMAPTRAV has sequential cohesion. It is activated to perform one of a set of operations on a single data structure, which is the higher, informational cohesion. To do the operation it performs a sequence of operations. It implements a state machine to perform the operations. Given the program is in a specific state, which state it goes to next is dictated by its input. The state machine is illustrated in Figure 19. The state and the input also determine the output in RESULT.

Variable	Meaning	Values
CUROP	State Variable	(Down, Across, Up, Stop)
STATUS	Input	(Attop, Empty, Atbot, Term, Endlev)
RESULT	Output	(Top, Newnode, Leaf)

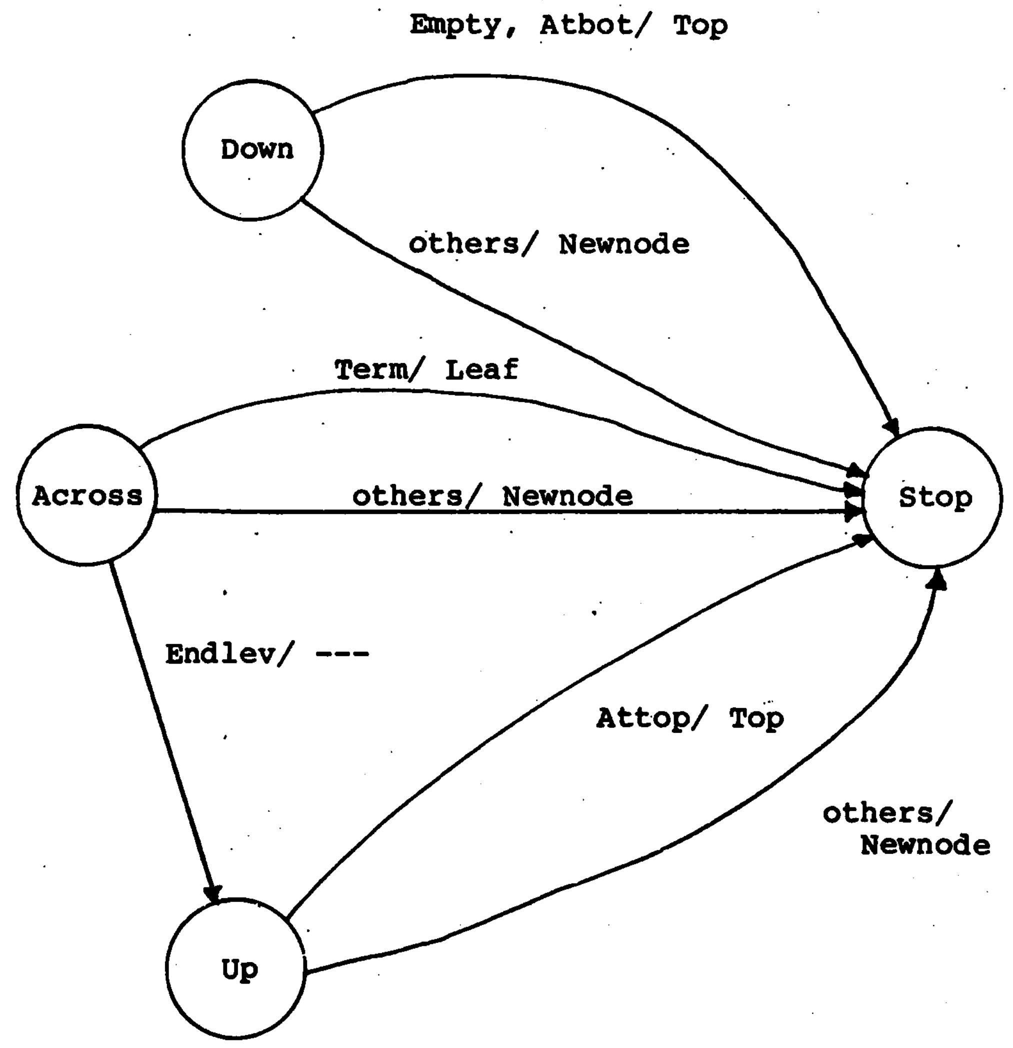


Fig 19. AMAPTRAV State Machine

AMOVE has strict informational cohesion. It implements the map move operations UP, DOWN, and ACROSS, and performs one of these each time it is called.

The modules each have other characteristics which aid in understanding and maintaining the code.

Module Code Characteristics. The modules were written to be easily debugged and maintained. Transportability to other computer systems was also a consideration. To achieve these goals all Pascal code was restricted to the statements found in Jensen and Wirth's PASCAL User's Manual and Report (Ref 14), and the source code was frequently clarified with comments for the reader.

One module, BITPU in the AUTIL1 library, is very inefficient in Pascal. Pascal does not contain commands to readily convert integers to bit patterns or bit patterns to integers. Using arithmetic operations is the only relatively non-machine dependent method of implementing the bit packing an unpacking needed. For specific applications this procedure should be implemented in the machine code for the computer used. BITPU is called for every node compare made by the FIND, DELETE, and RETRIEVE operations.

Summary

Implemented on a Model I TRS-80 personal computer in Pascal, the ADAM package is designed to be portable. The few machine dependent particulars are discussed, and the code is written to be easily translated to other programming languages.

ADAM's modularity gives the program package understandability, readability, and maintainability. With only a couple of exceptions, the ADAM modules have functional cohesion. The exceptions are well documented in the code.

Good software engineering, for quality assurance, required a user-friendly human interface. The interactive ADAMTEST package contains tests wherever possible to assure the user cannot easily get faulty inputs into the program system. The abundant, strategically placed input tests, available "HELP" listings, and meaningful prompts make ADAMTEST an interactive aid to learning the use of ADAM maps.

ADAM does contain some implementation dependent restrictions, however these are dependent on a maximum integer size of 16 bits. If the maximum integer size is smaller, the constants must be readjusted. However, if the maximum integer size is larger, they need not be changed.

V Analysis

Introduction

Some estimate of ADAM performance characteristics will be derived in this chapter. ADAM will be compared to other methods of data storage, mapping, and retrieval to determine the tradeoffs between methods. The ADAM map will be analyzed for its space requirements, and the ADD, DELETE, FIND, and RETRIEVAL operations for their time complexity. The structures to be compared for the above characteristics are

1. Linear unordered data sets, the benchmark;
2. Linear ordered data sets, the optimum 1-*dimensional* access structure, allowing simple binary searches on a single ordering;
3. Multiple dimension linear ordered data sets, a commonly used procedure for multikey access, and obtained by combining multiple attributes for a single unique key;
4. M-way tree forms in general with a comparison of
   - Fixed node sizes,
   - Binary trees in each node, the K-D tree,
   - Linear linked lists in each node, CARTAM and ADAM.

Some variables used in the analysis of the various access techniques are as follows:

- N number of data items in the data set
- K number of dimensions or associations shared by the data items in the data set
- L number of levels in the m-way tree major structure
- P number of bits of precision for the representations of the dimensions or associations
- s size of one storage unit, in bits
- R $=(K/e)^{(K+1/2)}$, the base of Stirling's bounds on K!, where e=2.71828.. (see CRC Standard Mathematical Tables, 14th Ed, 1965, page 433).

First the space requirements of each technique are considered.

Space Requirements

The space requirements of the different methods for storing data for associative access are analyzed for comparison, with a summary in Table III at the end of this section.

Linear Data Sets. Linear data sets need store only the K keys and the data item "value" in a sequential array in a random access form. Assuming each key to be P bits in length, and the data item "value" to be S_V in length, then the total size, S_T of a data set of N data items is

$$S_T = (KP+S_V)N \qquad \text{(V-1)}$$

and if S_K and S_V is a single storage unit of s bits, then

$$S_T = (KP+s)N \qquad (V-2)$$

bits. This is on the order of KPN, or O(KPN).

Multidimensional Linear Ordered Data Sets. Data sets with multidimensional linear orderings must have orderings for every combination of the K keys to be comparable to K-D trees, CARTAM, or ADAM. This gives K! orderings, which require at least K!-1 pointers for each data item. For homogeneity it is simpler to consider K! pointers, each of length S_p bits. Also, for each data item, the K keys of length P and the data item "value" of length S_V are also stored. These are stored for each of N data items, giving the total space requirement as

$$S_T = (K!S_P + KP + S_V)N \qquad (V-3)$$

If S_P, S_K and S_V each one storage unit, this gives

$$S_T = (KP + K!s + s)N \qquad (V-4)$$

storage units. Using Stirling's bounds on K!, this is O(RN).

M-way Tree Forms: Fixed Node Sizes. M-way trees with fixed node sizes include Quadtrees (Ref 5), which have K=2 keys, and B-trees (Ref 10:499), which can have any number of keys. Generally, the space requirements of these forms are optimized by point balancing the trees, but here region balanced trees are also considered, since they are similar to the ADAM map structure. Each node has 2^K pointers to represent K dimensional data and the key information to select the proper pointers, forcing the node size to be

$$S_N = 2^K S_P + KP + S_V \qquad \text{(V-5)}$$

For the point balanced m-way tree, N nodes are required, giving a space requirement of

$$S_T = N(2^K S_P + KP + S_V) \qquad \text{(V-6)}$$

which is $O(2^K N + NKP)$.

If the m-way tree is region-balanced, the number of levels is fixed at some value L. The number of nodes required is found by finding how many are needed to represent all the data points. The number of nodes at level i is given by

$$n_i = (2^K)^i - 1 \qquad \text{(V-7)}$$

where the root node is level i=1. Therefore, the number of levels, L, which may be required is given by the integer value of i which satisfies

$$n_i \geq N \geq n_i \qquad \text{(V-8)}$$

or

$$(2^K)^{L-1} - 1 \geq N \geq (2^K)^L - 1 \qquad \text{(V-9)}$$

Solving for L yields

$$\log(N+1) - K\log 2 \leq L \leq \log(N+1) - K\log 2 + 1 \qquad \text{(V-10)}$$

An estimate for L can be obtained by letting the base be 2^K, i.e.

$$\log(N+1) - 1 \leq L \leq \log(N+1) \tag{V-11}$$

The number of nodes required for L levels is given by

$$N_L = \sum_{i=1}^{L} (2^K)^{i-1} \tag{V-12}$$

$$= \frac{(2^K)^L - 1}{(2^K - 1)} \tag{V-13}$$

and, therefore, the space required is given by

$$S_T = 2^K(S_P + KP + S_V) \frac{(2^K)^L - 1}{(2^K - 1)} \tag{V-14}$$

Assuming that S_P and S_V are each one storage unit, then the space requirement is given by

$$S_T = \frac{(2^K)^{\log(N+1)} - 1}{(2^K - 1)} (2^K s + KP + s) \tag{V-15}$$

$$= \frac{N}{(2^K - 1)} (2^K s + KP + s) \tag{V-16}$$

The distribution of the N data points about the region dictate the number of nodes, and thus the total space required. If uniformly dispersed over the region, the maximum space required by N points is

$$S_1 = \frac{N}{(2^K - 1)} (2^K s + KP + s) \tag{V-17}$$

integer words for the "spread" portions of the tree which have all the branching to separate the nodes. This uses up L_1 of the L levels, where L_1 is the integer found from

$$L_1 -1 \leq \log N \leq L_1 \qquad \text{(V-18)}$$

The rest of the levels, $L-L_1$, have N nodes at each level, giving

$$S_2 = (L-L_1)N(2^K s + KP + s) \qquad \text{(V-19)}$$

storage units. The maximum number of storage units required for N data points in a region balanced tree is at most

$$S_{TMAX} = S_1+S_2 \qquad \text{(V-20)}$$

$$= \frac{N}{(2^K-1)} + (L-L_1)N(2^K s + KP + s) \qquad \text{(V-21)}$$

integer words, where L_1 is the largest integer such that $L_1 -1 \geq \log N$, and K is the number of keys.

The m-way tree with fixed node size storage requirement is then $O(LN2^K + LNKP)$, worst case.

<u>M-way Tree Forms: Variable Node Sizes</u>. The m-way trees with variable node sizes include region balanced K-D trees, CARTAM, and ADAM. Each of these forms uses a collection of smaller nodes to emulate a single variable sized m-way tree node. For the m-way tree node size calculations, there are some simplifications. The three implementations each form major tree structure m-way nodes of varying sizes.

For the region balanced K-D trees, the m-way node size varies from a minimum of

$$S_{KMIN} = KS_{KN} \tag{V-22}$$

where K is the number of keys and S_{KN} is the size of a K-D tree node, to

$$S_{KMAX} = (2^K-1)S_{KN} \tag{V-23}$$

When there is only one pointer out of the m-way node, the node size is S_{KMIN} because all K key decision levels must be represented in the node. When the node is full, S_{KMAX} is the resulting m-way node size. S_{KN} is two pointers, and the key may be explicit in the node's location, thus

$$S_{KN} = 2S_P \tag{V-24}$$

However, the key precision is P = L. Therefore, the K-D tree space requirements become

$$S_{TMAX} = \frac{N}{(2^K-1)} (2^K-1)S_{KN} + (L-L_1)NKS_{KN} \tag{V-25}$$

$$= NS_{KN}(1 + K(L-L_1)) \tag{V-26}$$

$$= 2NS_P(1 + K(L-L_1)) \tag{V-27}$$

This space requirement is linear in N, however, the maximum value is on the order of the product Of N, L, and K, or O(KNL).

For CARTAM, each m-way tree node consists of a linear linked list of between 1 and 2^K CARTAM nodes. The CARTAM nodes contain 2 pointers, K center values, and K delta values. The center values and delta values contain the key precision, and thus use P storage units each, and the pointers use S_P space. This gives the CARTAM node size as $S_{CN} = 2S_P + 2KP$. When no branches exist at a level, the nodes can be suppressed, giving the CARTAM space requirements as

$$S_T = \frac{N}{2^K-1} S^K S_{CN} \tag{V-28}$$

$$= \frac{N}{1-(1/2^K)} 2(S_P + KP) \tag{V-29}$$

This is O(NPK) space requirement.

ADAM packs the keys in each node to reduce the K keys to one K-bit field, giving the ADAM node size as

$$S_{AN} = 2S_P + K \tag{V-30}$$

ADAM cannot suppress non-branching levels as CARTAM does. The m-way tree nodes format consists of from 1 to 2^K ADAM nodes, but the node size is reduced to

$$S_{TMAX} = \frac{N}{2^K-1} 2^K S_{AN} + (L-L_1)NS_{AN} \tag{V-31}$$

$$= \left(\frac{N}{1-(1/2^K)} + (L-\log(N+1))N\right)(2S_P+K) \tag{V-32}$$

$$= (3s+K)\left(\frac{N}{1-(1/2^K)} + (L-\log(N+1))N\right) \tag{V-33}$$

Table III

Space Requirements

K = Number of Dimensions (Associations)
L = Number of Levels in Data Structure
N = Number of Data Items in the Data Set
P = Number of Bits of Precision
R = Stirling's Base for the K! Bound

Data Set Type	Maximum Space Requirement
Linear Data Set	O(KPN)*
Multidimensional Linear Ordered	O(RN)
M-way Tree -Fixed Node Size	$O(2^K N+NKP)$*
-Variable Node Size K-D Tree	O(LKN)**
CARTAM	O(KNP)
ADAM	O(NK(1+L+logN))**, ***

* This is a fixed value, not just a maximum.
** For these, P = L.
*** For $L-1 \leq \log N \leq L$ this is O(NK).

storage units. This is O(NK+LNK-NKlogN). For $L \doteq \log N$ this is O(NK), but L = P is neccessary to get the desired key precision, giving O(NK(1+P-logN)).

<u>Summary of Storage Requirements</u>. The space requirements of the given methods are summarized in Table III. The storage complexity of ADAM can be less than those of all the other techniques if the desired precision is approximately log base 2^K of N.

ADD and DELETE Time Requirements

Introduction. To analyze the time required for ADD and DELETE operations, the time to locate the data item position must be considered. Where necessary, the single data item find time is derived first, then the ADD and DELETE times are calculated.

Linear Unordered Data Set. A linear search must be performed to find a data item. However, no search is needed to ADD a data item. It can be ADDed to the bottom of the set in KP+s stores of one storage unit each, where store time is T_S, giving

$$T_A = (KP+s)T_S \tag{V-34}$$

which is O(KP) time.

To DELETE a data item, it must first be found via a linear search. The maximum search time is given by

$$T_{fMAX} = KPNT_C \tag{V-35}$$

where T_C is the compare time for one storage unit. All the keys of all N data items must be compared.

The DELETE time is then the time to find the data item plus the time to store a null for each key, or

$$T_D = T_f+KPT_S \tag{V-36}$$

which gives the maximum DELETE time as

$$T_{DMAX} = KPNT_C+KPT_S \tag{V-37}$$

T_{DMAX} is O(KPN).

Linear Ordered Data Set. Both ADD and DELETE first require a search to find where the value lies in the data set. Using the key sequence which is used for the ordering, a binary search finds the correct location in a maximum time of

$$T_{fMAX} = KPT_C\log N \tag{V-38}$$

and a minimum of $T_{fMIN} = KPT_C$ time.

Letting i be the location at which the data item is to be inserted, then the ADD time is given by

$$T_A = T_f+T_M+(KP+s)T_S \tag{V-39}$$

where T_M is the move time required to move all the data items from the ith to the Nth down one position to make room to insert the new data item. T_M varies from the time to move N data items, to the time to move none, or

$$0 \leq T_M \leq (KP+s)NT_S \tag{V-40}$$

Thus, the ADD time is

$$T_{AMAX} = KPT_C\log N+(KP+s)NT_S+(KP+s)T_S \tag{V-41}$$

$$= KPT_C\log N+(KP+s)(N+1)T_S \tag{V-42}$$

which is O(KPN).

The DELETE time consists of only the find time and the store time for null keys, and is given by

$$T_{DMAX} = KPT_C\log N+KPT_S \qquad (V-43)$$

which is O(KPlogN).

<u>Multidimensional</u> <u>Linear</u> <u>Ordered</u> <u>Data</u> <u>Sets</u>. To ADD and DELETE a data item using multidimensional linear ordered data sets requires finding the data item in each of the K! orderings. This requires a maximum search time of

$$T_{fMAX} = K!KPT_C\log N \qquad (V-44)$$

followed by the ADD or DELETE operation.

The ADD operation requires moves in each of the K! orderings which vary as in the single linear ordering

$$0 \leq T_M \leq (KP+s)NT_S \qquad (V-45)$$

giving the total ADD time a maximum value of

$$T_{AMAX} = K!(KPT_C\log N+(KP+s)NT_S+T_S)+(KP+s)T_S \qquad (V-46)$$

which is O(KRPN).

The DELETE operation requires only storing null keys and/or pointers. To store both null keys and pointers requires

$$T_{DMAX} = K!(KPT_C\log N+KPT_S)+KPT_S \qquad (V-47)$$

maximum which is O(RPKlogN).

However, to store only null key values requires a maximum time of

$$T_{DMAX} = KPT_C\log N + KPT_S \qquad (V-48)$$

or O(KPlogN) time.

This derivation assumes that only one ordering needs to be searched to find the data item and its key values.

<u>M-way Tree Forms: Fixed Node Sizes</u>. To ADD and DELETE within all m-way tree forms, the data item position in the tree must first be located. After it is located, the ADD or DELETE operation may be performed.

In general, for m-way trees, there is a node search time to find which branch to take. This node search time is designated T_{FN}.

To find a data item, a sequence of nodes must be searched, one for each level in the tree. For point balanced trees, the number of levels is given by $L-1 \leq \log(N+1)$, where L is the largest integer which satisfies. For L levels, the general m-way tree find time is

$$T_F = LKPT_R \qquad (V-49)$$

For fixed node sizes, where the subtree pointers are ordered and selected by index. T_R includes the time to retrieve K keys and hash to the proper pointer at each of the L levels.

To ADD data items requires finding the location and adding the necessary pointers and nodes. The search will leave the user at some level, L_1, such that $L_1 \leq L$, and then to ADD will require adding a pointer, then rebalancing the tree. The pointer is added in fixed time T_S, and the balancing is done in time T_B. The balance time will not be further analyzed as it can be substantial, and complex in large trees. The ADD time is given by

$$T_A = L_1KPT_R+T_S+T_B \qquad \text{(V-50)}$$

This can be misleading if it is not realized that T_B can be a function of K, P, N, and L. This formulation is not easily compared to ADD times of other forms.

To DELETE data items can give a similar result to the ADD operation,

$$T_D = LKPT_R+T_S+T_B \qquad \text{(V-51)}$$

but also is not easily compared to other forms.

Region-balanced m-way trees are somewhat easier to analyze for comparison. No point balancing is needed after ADD's and DELETE's, and L is fixed, giving an ADD time of

$$T_A = L_1KPT_R+(L-L_1)T_S \qquad \text{(V-52)}$$

The structure, must be created from the last match at level L_1, to level L. The DELETE time is given by

$$T_D = LKPT_R+T_S \qquad \text{(V-53)}$$

<u>M-way Tree Forms: Variable Node Sizes</u>. For all m-way trees, the time to find where to ADD or DELETE a data item is given by

$$T_f = LT_{fN} \qquad \text{(V-54)}$$

However, T_{fN} is a function of m-way tree node size when the node size varies, so a more accurate measure of search time is

$$T_f = \sum_{i=1}^{L} T_{fNi} \qquad \text{(V-55)}$$

The node search time, T_{fNi}, can be estimated for the different structures. For all the m-way trees, the addition or removal of the data item is the only other operation to perform in the ADD and DELETE operation sequence.

For region balanced K-D trees, the ADD time is dependent on the search time at each m-way tree node T_{fNi}, the number of levels searched, L_1, and the number of levels added to ADD the data item. The node search time is

$$T_{fNi} = KT_C \qquad \text{(V-56)}$$

where T_C is the time for one compare. This gives the ADD time for the K-D tree as

$$T_A = \sum_{i=1}^{L_1} KT_C \; +(L-L_1)(2sT_S+T_G) \qquad \text{(V-57)}$$

where $2sT_S$ is the time to store 2 pointers, and the key is implicit, requiring no time. T_G is the time to get a free node. This simplifies further, since there is only one path per level, to

$$T_A = L_1KT_C+(L-L_1)(2sT_S+T_G) \qquad \text{(V-58)}$$

which is O(LK).

For the CARTAM structure, each m-way node requires $1<n_i<2^K$ CARTAM node compares, T_{CN}. Each CARTAM node compare requires 2K key precision compares or

$$T_{CN} = 2KPT_C \qquad \text{(V-59)}$$

and the ADD time is

$$T_A = \sum_{i=1}^{L_1} n_i(2KPT_C) \ +(2KP+2s)T_S+T_G \qquad \text{(V-60)}$$

$$= 2KPT_C \sum_{i=1}^{L_1} n_i \ +2(KP+s)T_S+T_G \qquad \text{(V-61)}$$

This can vary from $n_i=1$ to $n_i=2^K$ for all i, yielding

$$T_{AMAX} = 2KPT_CL_12^K+2(KP+s)T_S+T_G \qquad \text{(V-62)}$$

which is $O(KP2^KL)$.

ADAM requires $1\leq n_i\leq 2^K$ ADAM node compares. Each ADAM node requires K compares, each of a single bit requiring T_C time. However, ADAM requires a repacking of the keys from K keys to L level search keys adding $(KP+LK)T_B$ for packing and unpacking bits. These give the ADD time as

$$T_{AMAX} = L_1K2^KT_C+(KP+LK)T_B+(L-L_1)((2s+K)T_S+T_G) \qquad (V-63)$$

which is $O(LK2^K)$.

For DELETE times, the insert times for the added nodes can be ignored and for K-D trees the result is

$$T_D = LKT_C2^K+2sT_S+T_R \qquad (V-64)$$

where 2 pointers are stored and the removed portion is returned to free storage. This is $O(LK)$.

CARTAM DELETE time is given by

$$T_{DMAX} = 2KPT_CL2^K+2PT_S+T_R \qquad (V-65)$$

which is $O(KPL2^K)$.

ADAM DELETE time for a single data item is given by

$$T_{DMAX} = LK2^KT_C+(KP+LK)T_P+2sT_S+T_R \qquad (V-66)$$

which is $O(LK2^K)$.

ADAM, however, has a region delete feature, where a single DELETE can remove all data items within a previously found region. The time requirements for this is the time required to traverse all subtrees which contain any points within the region. If an entire subtree is within the region to be DELETE'd, the subtree is removed by storing only 2 pointers, however, the entire subtree must then be returned to free storage. This DELETE time for N_D data items can be estimated by

$$T_{DMULT} = T_{DMAX} + MT_F \qquad \text{(V-67)}$$

which is depends on the ADAM FIND time, and the number of FINDs which are performed.

Summary of ADD and DELETE Times. The ADD and DELETE times for the various methods are summarized in Table IV. To DELETE whole regions of data points in one operation in all methods requires merely a more complex compare operation, and thus a longer compare time. To DELETE an entire region of n points does not take as much time as n single data item deletions if the algorithms are adjusted for region compares rather than single point values, as in ADAM. Point retrieval can be based on the same concepts, by using either single points or by using all points within a region.

Table IV

ADD and DELETE Time Requirements

K = Number of Dimensions (Associations)
L = Number of Levels in Data Structure
N = Number of Data Items in the Data Set
P = Number of Bits of Precision
R = Stirling's Base for K! Bounds

Data Set Type	Maximum ADD Time	Maximum DELETE Time
Linear Unordered	O(KP)	O(KPN)
Linear Ordered	O(KPN)	O(KPlogN)
Multidimensional Linear Ordered	O(RPN)	O(KPlogN)
M-way Trees -Variable Node Size K-D Tree	O(KL) *	O(KL) *
CARTAM	$O(KP2^KL)$	$O(KP2^KL)$
ADAM	$O(K2^KL)$ *	$O(K2^KL)$ *

* For these L = P.

FIND and RETRIEVE Time Requirements

Introduction. Generally FIND is part of both the DELETE and RETRIEVE operations. However, in ADAM they can be separated, and multiple FIND operations can be performed before a single DELETE or RETRIEVE is done. This allows building regions of data for RETRIEVE or DELETE operatons which are the UNION of several simple FIND regions. Alternate ways of performing this compound FIND, to be followed by a RETRIEVE or DELETE operation, on other than ADAM data sets, are as follows:

1. Perform multiple DELETE or RETRIEVE operations to eventually cover the entire region; or
2. Perform a complex region compare, instead of the normal interval compare, for each compare that would otherwise be done.

The first method is easy and places the burden of irregularly shaped region handling on the user. The second may require a fairly complex region description technique and comparison algorithm. For this analysis, all region retrievals will be rectangular regions consisting of a single interval in each dimension. CARTAM uses a Cartesian measure to retrieve circles or spheres of data, which are not easy to analyze, so the CARTAM structure will be analyzed for only rectangular regions. The FIND and RETRIEVE operations will be considered as a single FIND operation for all the data forms except ADAM.

Linear Unordered Data Sets. To FIND a rectangular region in K dimensions in a linear unordered data set requires 2 compares on each key for each data item in the set. This requires a FIND time of

$$T_F = 2KPNT_C + N_F T_R \qquad \text{(V-68)}$$

where $N_F T_R$ is the retrieval time for the N_F data items found, requiring T_R retrieval time each. T_F is then O(KPN).

Linear Ordered Data Sets. The ordering of a linear ordered data set can be used only to reduce the N data items to some number $N_1 \leq N$. To do this, the two extreme points of the region, under the ordering of the data set, may be found using a binary search. These extreme points then bound an interval of N_1 data items, which are not ordered properly for further single key-sequence searches, thus must be linearly searched. This generates

$$T_F = 2KP\log N\ T_C + 2KPN_1 T_C + N_F T_R \qquad \text{(V-69)}$$

$$= 2KPT_C(\log N + N_1) + N_F T_R \qquad \text{(V-70)}$$

which is $O(KP\log N + KPN_1)$. The relative sizes of logN and N_1 determine which dominates the expression. N_1 can vary from O(N), for retrievals of most of the original data set, to O(1), when only a small fixed number of data points are retrieved. The maximum time is

$$T_{FMAX} = O(KP\log N + KPN) \qquad \text{(V-71)}$$

$$= O(KPN) \qquad \text{(V-72)}$$

for the linear ordered data set.

__Multidimensional__ __Linear__ __Ordered__ __Data__ __Set__. For multiple linear orderings, the search can be on one ordering, as in the linear ordered data set, or it can be on more of the orderings. Assuming K keys, K orderings will be used.

First, the binary search time for the K orderings is calculated. The K orderings use the binary search to locate the interval, and each of the K intervals may require further searching. Further searching may be replaced by an intersection operation on these K interval sets of data items. The K sets each have n_i elements each, for $i=1,\ldots,K$. Since the sets are ordered using different key sequences, the intersection set, of n_{ab} data items, is derived from sets containing n_a and n_b data items each, and requires $n_a n_b$ compares. To intersect all K sets, one can perform a sequence of set intersections, each intersection using a new set and the resulting set from the previous intersection giving

$$T_I = KPT_C \sum_{i=1}^{K-1} n_{i+1} N_i \tag{V-73}$$

where N_i is the number of elements left after the (i-1) intersection, and $N_1=n_1$. The N_i values must form a nonincreasing nonnegative sequence, such that $0 \le N_{K-1} \le \ldots \le N_1 \le N$. The extremes are $N_i=0$ for all $i=1,\ldots, k$, and $N_i=N$ for all $i=1, \ldots, K$. This gives

$$T_{IMAX} = KPT_C \sum_{i=1}^{K-1} n_{i+1} N \qquad \text{(V-74)}$$

But, for N_i, $i>1$, $n_i \geq N_i$ and for $N_i=N$ for all $i=1, \ldots, K$, $n_i=N$ for all $i=1, \ldots, K$. Thus,

$$T_{IMAX} = KPT_C \sum_{i=1}^{K-1} N^2 \qquad \text{(V-75)}$$

$$= KPT_C(K-1)N^2 \qquad \text{(V-76)}$$

$$= (K^2-K)PN^2T_C \qquad \text{(V-77)}$$

which is $O(K^2PN^2)$.

To get the FIND time for a region, then

$$T_F = (2KPT_C \log N)K+T_I \qquad \text{(V-78)}$$

since only K of the K! orderings were used. This gives

$$T_{FMAX} = 2K^2PT_C \log N+(K^2-K)PN^2T_C \qquad \text{(V-79)}$$

which is $O(K^2PN^2)$, for $N_F=N$. However, if one retrieval comes back with an empty interval, then no further compares need be performed.

For the multiple linear ordered data set, the search via a single ordering, starting with a binary search, followed by a linear search of N_1 data items, will result in

$$T_F = 2KPT_C(\log N+N_1)+N_FT_R \qquad \text{(V-80)}$$

which is $O(KPlogN+KPN_1)$. This gives the same results as for the linear ordered data set, $T_{FMAX}= O(KN)$, for the multidimensional linear ordered data set.

<u>M-way Tree Forms: Fixed Node Sizes</u>. The general m-way tree consists of levels of nodes. Each node is the root of a subtree, and that subtree includes every data item within a specific subregion. The subregion of each node at a specific level has an empty intersection with the subregions of each other node at that level. The pointers in a node compose the branches of the subtree rooted at that node.

To find the subtrees of a node which lie within a region, compares must be performed between the search region and the region represented by each subtree present. This requires $2K2^K$ compares, or 2K compares for each pointer. The m-way tree node compare time, T_{NC}, is

$$T_{NC} = 2K2^K PT_C \tag{V-81}$$

for a node of size 2^K pointers. Each subtree compare returns 3 cases as follows:

1. Subtree region inside search region,
2. Subtree region outside search region, and
3. Subtree region overlaps search region.

For case 1 and 3 the subtree needs to be traversed. For case 1 the entire subtree can be retrieved without any further compares, and for case 2 the subtree can be eliminated without any further compares. The resulting time of access on any given tree then depends on the number of nodes which are on the border of the region, N_B, and number of nodes within the region, N_W.

For the ith border node, i=1,..., N_B, the compare time is T_{NCi} and for the jth included node, j=1, ..., N_W, the traversal time is T_{TNj}. The tree FIND time for the situation is

$$T_F = \sum_{i=1}^{N_B} T_{NCi} + \sum_{j=1}^{N_W} T_{TNj} \qquad (V-82)$$

The node traversal time T_{TNj} is fixed for 2^K pointers, assuming traversal over a null pointer takes as long as traversal over a non-null pointer. This yields

$$T_{TNj} = 2^K T_T \qquad (V-83)$$

and

$$T_F = \sum_{i=1}^{N_B} T_{NCi} \quad + 2^K N_W T_T \qquad (V-84)$$

From this comes a worst case, with $T_{NCi} = 2K2^K PT_C$ and for all N_B,

$$N_B = \frac{N}{2^K - 1} \qquad (V-85)$$

and $N_W = N$. This gives

$$T_{FMAX} = N_B(2K2^K PT_C)+2^K N_W T_T \tag{V-86}$$

$$= 2KP\frac{N}{(1-(1/2^K)} T_C+2^K N_W T_T \tag{V-87}$$

which is O(KPN) for the m-way tree with a fixed node size.

<u>M-way Tree Forms:</u> <u>Variable Node Sizes</u>. The variable node size forms all have some number of nodes, n_i, for each m-way tree node with

$$0 < n_i \leq 2^K \tag{V-88}$$

The K-D tree has $K \leq n_i \leq 2^K-1$ nodes per m-way tree node, thus, the comparison time for a single m-way tree node is given by

$$T_{NCi} = n_i T_C \tag{V-89}$$

which varies from

$$T_{NCMAX} = (2^K-1)T_C \tag{V-90}$$

to

$$T_{NCMIN} = KT_C \tag{V-91}$$

Using $n_{MAX}=(2^K-1)$ and the value for T_{NCMAX} above, the FIND time is given by

$$T_F = T_C \sum_{i=1}^{N_B} n_i +T_T\sum_{i=1}^{N_W} n_i \tag{V-92}$$

which, for

$$N_B = \frac{N}{2^K-1} \tag{V-93}$$

and N_W=N, gives

$$T_{FMAX} = T_C \frac{N}{2^K-1}(2^K-1)+T_T N(2^K-1) \tag{V-94}$$

$$= NT_C+2^K NT_T \tag{V-95}$$

which is $O(N2^K)$ for the K-D tree.

CARTAM trees have $1 \leq n_i \leq 2^K$ nodes per m-way tree node, and thus, T_{NC} varies from

$$T_{NCMAX} = 2KP2^K T_C \tag{V-96}$$

to

$$T_{NCMIN} = 2KPT_C \tag{V-97}$$

These equations give the FIND time maximum as

$$N_B = \frac{N}{2^K-1} \tag{V-98}$$

and N_W=N, giving

$$T_{FMAX} = 2KPT_C \frac{N}{2^K-1} 2^K+T_T N2^K \tag{V-99}$$

$$= 2KPT_C \frac{N}{1-(1/2^K)} +T_T 2^K N \tag{V-100}$$

which is $O(NKP+N2^K)$ for the CARTAM tree structure.

ADAM trees are similar to CARTAM for the RETRIEVE, but, ADAM nodes contain only the K bits for its keys, rather than the 2^K keys of P bits each used by CARTAM. ADAM uses these K bits as its key field in each ADAM node. This gives

$$T_{NCMAX} = K2^K T_C \qquad \text{(V-101)}$$

and a total RETRIEVE time of

$$T_{FRMAX} = KT_C \frac{N}{1-(1/2^K)} + T_T 2^K N \qquad \text{(V-102)}$$

which is $O(KN+2^KN)$, or $O(2^KN)$. However, ADAM has a different value for its FIND function.

For the FIND operation, one need only traverse subtrees which consist of the N_B nodes which define the borders of the search region. This gives ADAM the FIND time of

$$T_F = \sum_{i=1}^{N_B} T_{NCi} \qquad \text{(V-103)}$$

$$= KT_C \sum_{i=1}^{N_B} N_i \qquad \text{(V-104)}$$

But, $1 \leq n_i \leq 2^K$, thus, for

$$N_B = \frac{N}{2^K-1} \qquad \text{V-105)}$$

then

$$T_{FMAX} = KT_C \frac{N}{1-(1/2^K)} \qquad \text{(V-106)}$$

which is O(KN) for ADAM using the FIND operation only.

ADAM can perform multiple FIND operations in less time than multiple RETRIEVE operations. To retrieve complex regions, multiple FIND operations followed by a single RETRIEVE can be faster than multiple RETRIEVE operations.

Summary of FIND and RETRIEVE Times. The FIND and RETRIEVE operation times are summarized in Table V. All the FIND operations are limited by compares that can be done with rectangular or convex regions except ADAM which can utilize successive FIND operations to define a complex, convex shaped region with a concave surface, disjoint subregions, or any other describable region form in a single RETRIEVE.

Table V

FIND and RETRIEVE Time Requirements

K = Number of Dimensions (Associations)
N = Number of Data Items in the Data Set
P = Number of Bits of Precision

Data Set Type	Non-primary Key Search Maximum Time
Linear Unordered	O(KPN)
Linear Ordered	O(KPN)
Multidimensional Linear Ordered	O(KPN)
M-way Tree -Fixed Node Size	O(KPN)
-Variable Node Size K-D Tree	$O(2^K N)$
CARTAM	$O(NKP+2^K N)$
ADAM FIND	O(KN)
RETRIEVE	$O(2^K N)$

Table VI

Performance Ranking of Techniques

Data Set Type	Space	ADD	DELETE	FIND
Linear Unordered	1-5	1,2	6	2-4
Linear Ordered	1-5	5	2,3	2-4
Multidimensional Linear Ordered	6	6	2,3	2-4
M-way Trees -Variable Node Size K-D Tree	1-5	1,2	1	5,6
CARTAM	1-5	4	5	5,6
ADAM	1-5	3	4	1 *

* ADAM retrieves an associative data set, while all the others retrieve sequential data sets. Retrieving a sequential data set, ADAM is comparable to K-D trees.

ADAM Performance Optimization

Table VI summarizes the relative performance of the different methods which currently are used in associative access applications, and ADAM. By selecting ranges of values for the precision, P; the number of levels in the tree, L; the number of dimensions, K; and the number of retrieval regions, M; all relative to the other parameters, ADAM can be shown to have a performance exceeding all other methods in most operation. These ranges can be found for large values of N, the number of data items.

These parameter ranges can be set by checking each category of analysis, space, ADD time, DELETE time, and FIND/RETRIEVE time.

Space. For L = log N ADAM uses the least space. This is because ADAM takes advanatage of redundant significant lists in the values. For K and P such 2^K > KP the m-way tree with fixed node size is worse than linear data sets (see Table III), but for 2^K < KP it is tied with them.

For all the region-balanced m-way tree forms with variable node size the order of space requirement is the same as for the linear data set. However, for L = log N ADAM is reduced to O(NK) and is more space efficient than any of the other techniques.

ADD Time. ADAM came in third for ADD time. The 2^K compares possible at each level make ADAM worse than a linear unordered or the K-D tree (see Table IV), but for 2^K < N ADAM is still better than linear ordered or CARTAM.

DELETE Time. Here ADAM came in fourth. The single data item search time with 2^K compares at each level made the difference again. However, if 2^K is less than log N then ADAM can be in second place (see Table IV), behind only the K-D trees. If ADAM is called upon to delete a number of regions, it can delete them all in a single DELETE operation, by applying multiple, faster, FIND operations to define the regions to DELETE first, then deleting them all in one operation.

FIND and RETRIEVE Time. The forte of the ADAM technique is its FIND operation. The ADAM FIND operation allows irregularly shaped regions to be easily defined by multiple rectangular retrieval regions, and through the use of multiple FIND operations, defines the multiple rectangles as a single region. This region can then be deleted or retrieved in a single operation. For the purpose of this project, CARTAM was analyzed for rectangular region retrievals, however it actually uses circular or spherical regions. Since multiple circular and spherical regions don't efficiently cover a region without overlap, rectangular regions were used. The multiple FIND operations of the ADAM technique, followed by a single RETRIEVE or DELETE, replace the multiple FINDs or DELETEs required by each of the other techniques.

Each of the methods could be implemented to use other than rectangular search regions, however this would have to be a much less generalized implementation than is defined for this project.

The ADAM, FIND operation retrieves an associative data set, and is the best performer here. One may argue that P bits of precision are one compare, giving O(KN) in the linear methods, but examples to the contrary are 32-bit values on an eight-bit machine, or 512 digit numbers on any modern machine. Likewise, for small values of K, ADAM can perform one compare for all K fields at each node, giving O(N) performanace, however, for large enough values of K, multiple compares may be needed.

For actual comparison of ADAM to the other methods a retrieval of sequential data set may be a good example. For a region retrieval, where the region is composed of M rectangular regions, the other methods all require multiplying the retrieval time for the method by M. However, for ADAM it becomes

$$T_F = M\ O(KN) + O(2^K N) \qquad \text{(V-107)}$$

$$= O(MKN + 2^K N) \qquad \text{(V-108)}$$

$$= O(N(MK + 2^K)) \qquad \text{(V-109)}$$

for $MK + 2^K < MKP$ ADAM is still more efficient for the retrieval of a sequential data set.

Overall. To get the best "worst case" performance from ADAM, specific parameters must be considered, i.e.

- Space, $p = L = \log N$
- ADD time, $2^K < N$
- DELETE time, $2^K < \log N$
- Sequential RETRIEVE time, $M > \frac{2^K}{K}$

These restrictions put ADAM first in space requirements and sequential data set time requirements, and second only to K-D trees in ADD and DELETE time requirements.

Run Time Performance Analysis

Project time constraints prevented the installation of timing measurement instrumentation into the ADAMTEST program package. The package could have contained start time and stop time checkpoints, and allowed all operations to be timed after their initiation. These start and stop time values could then have been compared to a comparable set of operations using some of the other techniques. Each technique would have been implemented with a similar top level interface such as ADAMTEST. Allowances also should be made for multiple operations such as a series of data ADDs, DELETEs, and RETRIEVEs. These operations could then have been timed and compared in an actual working environment, but the environment would be have been standardized enough to eliminate differences caused by different implementation techniques.

The suggested techniques to implement for comparison are the linear ordered data set and the K-D tree data set. The linear ordered data set is the commonly used "milestone" for comparison, and the K-D tree is the most recently acclaimed associative access technique. Using real time test runs on these techniques could have given a realistic way to validate the results of the theoretical analysis.

Analysis Summary

The theoretical analysis of the different techniques for performing associative access to data gives a clear view of some of the problems with those methods. Each technique has advantages and its disadvantages, and only a technique which has been picked for a specific application can be guaranteed to be a good performer. ADAM performs well when applied to a particular combination of problems present in very large databases. ADAM especially appears to be well suited to the case where the keys are very long, although more than just a few keys can seriously degrade its performance.

Much more analysis is needed in the area of irregularly shaped or disjoint region retrieval. Also, some run time analysis is needed. Many possibilities for application exist, and each application needs a comparison analysis to choose among these techniques.

VI Results

Introduction

ADAM models homogeneous real world data associations. This chapter covers the model characteristics as described from Chapter II and III. The implementation code characteristics are then mentioned, as well as the data flow diagrams in Appendix B, the Warnier-Orr data structure charts in Appendix C, and the complete ADAM program package source listing in Appendix D. Theoretical performance of the model implementation, from the analysis in Chapter V is then discussed. Finally, the functional tests performed on ADAM are discussed, with references to the test run listings in Appendix E.

Data Model Characteristics

The ADAM data model is derived in Chapter II, and refined in Chapter III. The model is restricted to data representable as having K homogeneous associations, where $1 \leq K$. To represent an association in the ADAM model, the associatiton must be mapped to numerical values in the half open interval [0,1). The user then determines the data resolution desired, and sets the level number to the number of bits required for that resolution. Once the data is stored in the ADAM association model, the data can be retrieved on any combination of association value, or key, intervals.

To retrieve an irregularly shaped region the user merely defines multiple rectangular regions to approximate the region shape, then retrieves the entire region in one RETRIEVE operation. Region definitions can be made independent of a single dimension, or association, by specifying the range for that dimension as [0,1].

The ADAM data model allows the definition and retrieval of multiple data items by region. The regions can be any union of rectangular retrieval regions, thus allowing disjoint and irregularly shaped retrieval regions. Chapter III and IV give the design and describe the implementation. The code is dicussed next.

Code Characteristics

The more than 1600 lines of code in the ADAM program package are highly structured. Following the requirements set forth in Chapter II for quality assurance, ADAM was designed and implemented in a generally top-down fashion. Some anticipated low-level functions are exceptions to the top-down implementation. This structure is shown by the data flow diagrams of Appendix B. Strict modularity was enforced, and is discussed further in Chapter IV in the section "Program Modules". The needed data structures were defined for clarity using the Warnier-Orr data structure diagrams in Appendix C. The full source code is given in Appendix D.

Machine dependent and language dependent features were avoided. Though Pascal was the language used, recursive procedure calls, pointer variables, and "hidden storage" management procedures were avoided to allow easy translation of the programs to FORTRAN or other medium to low order languages.

The implementation was designed for easy functional testing and algorithm debugging. For this reason, the total ADAM map buffer size was restricted to 50 nodes. This allows about 10 data items in a 6 level structure. The implementation also limits the number of dimensions to $1 \leq K \leq 10$, the number of regions to $0 \leq R \leq 8$, and the number of levels to $3 \leq L \leq 16$. For general applications these limits may be easily adjusted.

All user inputs are tested to assure they are within the legal limits. Illegal inputs are either ignored, with the prompt repeated, or used as a signal to abort a portion of an operation. The checking and "HELP" command list combine to make the package a useful training aid for teaching the use and interpretation of the ADAM model. Although the program package does not contain instrumentation for evaluating performance times, The performance times were estimated and compared to other access techniques.

Theoretical Performance

In Chapter V, different techniques for associative data retrieval are analyzed for storage space, ADD time, DELETE time, and FIND time requirements. ADAM was shown to be the most space efficient of the methods if the number of levels is chosen properly. ADAM ADD and DELETE times were good, but not the best, and ADAM's FIND time was outstanding. The drawback is that the ADAM FIND retrieves an associative data set which must be translated to a sequential data set for use. However, this translation is done by a RETRIEVE operation which is comparable in speed to the FIND operations of the best of the other access techniques.

The theoretical analysis makes ADAM appear extremely fast and storage efficient for large amounts of data. However, due to time constraints, timing instrumentation and alternate access techniques were not implemented for comparison. The code, however, was written for ease of maintenance and upgrade, so instrumentation could be added. The program package is interactive, and includes a DEBUG package which allows complete functional testing.

Functional Tests

To debug and maintain the software package, a DEBUG library was included in the ADAM package. The DEBUG operations allow a formatted dump of the ADAM map nodes. Using the DEBUG features, a complete functional test sequence is given in Appendix E. The operations ACREATE, AADD, ARETRIEVE, AFIND, and ADELETE were each tested. As the map structure was modified it was dumped to print to allow validation of all the map node's contents.

Summary

Theoretically, the ADAM package should exhibit very high performance quality. It is the most space-efficient of the techniques analyzed in Chapter V, and it has the shortest FIND time. Functionally it works exactly as specified in Chapter III. However, no time performance comparisons have been made.

The software packages use software engineering practices throughout to help guarantee quality as required in Chapter II. The 1600 plus lines of code are modular, well-commented, and otherwise fully documented in this thesis. They are also written for human interface, including input testing, meaningful prompts, and "HELP" displays.

The data representation and access may be unfamiliar to new users, and thus, the software package generated can serve as an introductory learning tool, allowing the user to later apply the ADAM algorithm set to specific applications. ADAM is a revolutionary concept for representing and accessing data.

The ADAM concept is a valid solution to the secondary key access problem.

VII Conclusions and Recommendations

Conclusions

ADAM appears to be the basis for a new approach to data access. It consists of good concepts from several other techniques, combined in a new data representation. The data representation and its associated algorithm set is incorporated into a high quality software package. The result is a versatile solution to the multikey retrieval problem.

The ADAM map derives its overall structure from CARTAM, but changes the tree structure's key form. The result is a new data representation. The ADAM algorithm set is defined to "hide" the actual implementation from the user. The resulting associative access method gives both space efficient data storage, and rapid data retrieval.

The ADAM software routine package was generated using the highest quality software engineering techniques. These include rigid top down structure, modularity, and quality documentation. These combine to make the package easily understood and maintained, as well as making the package easy to use for varied applications.

The ADAM package gives a quality solution to the multikey access problem. The result gives easy access to data via data item key match queries, key range queries, and, something not common in current literature, multidimensional key region queries.

ADAM has the potential for widespread use in database access, computer graphics, and pattern recognition. However each of these areas of possible application need researched further.

Recommendations

The ADAM package needs further analysis. Some of the areas needing further investigation include ways to make it handle large volumes of data and multiple simultaneous users by implementing

- ADAM map checkpoint and restore operations,
- Virtual index access to the ADAM map buffer, and
- Parasitic map retrieval operations.

These are necessary for large database applications. Also, a generalized region compare operation should be investigated for applications in pattern recognition.

ADAM requires further analysis relating performance to data set parameter values such as number of attributes per data item, bits of precision per attribute, and data items per data set. Also ADAM's sensitivity to data point distribution patterns within the data region should be investigated. Parameter combinations and point distribution patterns which easily lend themselves to the ADAM structure should be analyzed in depth. Specific criteria should be established for when and where an ADAM technique should be applied rather than one of the other available access techniques. Analysis should also include actual run time comparisons.

Some capabilities should be added to the basic ADAM algorithm set, along with the necessary map modifications. These are as follows:

-Parasitic Map access

-Generalized region compare

-Disk save/restore for ADAM maps

-Virtual map buffer pointers and indexing

Parasitic maps should be implelmented for a multiuser environment. These should be analyzed for performance, maintainance or database integrity, and speed of access.

Before ADAM maps can be made very large, disk save/restore routines should be implemented to checkpoint and restore the maps. Also, for limited memory machines a form of virtual indexes to access data address spaces exceeding the user's computer memory size should be considered.

The generalized region compare is a pattern recognition feature of ADAM. The user predefines region "categories" in the database, then the ADAM map can be accessed via a single data item, returning a region indicator. This would allow a user to have a "find category" query for data points. A fast query of this sort is a new approach to the decision functions discussed in Nilsson's Learning Machines (Ref. 22).

The availability of rapid access to variously shaped regions using ADAM, makes ADAM a candidate for graphic scene generation from 3-dimensional "point" models. This capability of ADAM should be investigated further for application in projects such as aircraft cockpit simulators, robot environment modelling, and design automation.

BIBLIOGRAPHY

1. Aho, A. V., J. E. Hopcraft, and J. D. Ullman. The Design and Analysis of Computer Algorithms. Reading, Massachusetts: Addison-Wesley Publishing Co. 1974.

2. Baer, J. Computer Systems Architecture. Rockville, Maryland: Computer Science Press, Inc., 1980.

3. Bentley, J. L. "Multidimensional Binary Search Trees Used for Associative Searching," Communications of the ACM 18, (September 1975).

4. ----- "Multidimensional Divide and Conquer." Communications of the ACM: 214-229 (Apr 1980).

5. Bentley, J. L. and R. A. Finkel. "Quad Trees: A Data Structure for Retrieval on Composite Keys," Acta Informatica 4: 1-9 (1974).

6. Bentley, J. L. and J. H. Friedman. "Data Structures for Range Searching," ACM Computing Surveys, 11: 397-409 (December 1979).

7. Chistofides, N. Graph Theory: An Algorithmic Approach. New York, New York: Academic Press, 1975.

8. Comfort, D. and W. Erickson. Relational Information Management System (RIM). Computer program source listing. Seattle, Washington: Boeing Commercial Airplane Co., January 1981.

9. Date, C. J. An Introduction to Database Systems. Third Ed., Reading, Massachussetts: Addison-Wesley Publishing Co. 1981.

10. Horowitz, E. and S. Sahni. Fundamentals of Data Structures. Potomac, Maryland: Computer Science Press Inc. 1979.

11. Hunter, G. M. and K. Steiglitz. "Operations on Images Using Quadtrees," IEEE Transactions on Pattern Analysis and Machine Intelligence, 1: 145-153 (January 1979).

12. ----- "Linear Transformations of Pictures Represented by Quadtrees," Computer Graphics and Image Processing, 10: 289-296 (1979).

13. Ichikawa, T., Y. Tajima, and M. Yamamura. "Retrieval of Image Features in Terms of Content-Addressing of Hierarchically Structured Image Data," Paper presented at the IEEE Computer Society Workshop on Picture Data

Descriptions, 1980.

14. Jensen, K. and N. Wirth. PASCAL User's Manual and Report. Second Ed., New York, New York: Springer-Verlag 1975.

15. Knuth, D. E. The Art of Computer Programming Vol 1: Fundamental Algorithms. Reading, Massachusetts: Addison-Wesley Publishing Co. 1968.

16. ----- The Art of Computer Programming Vol 3: Sorting and Searching. Reading, Massachusetts: Addison-Wesley Publishing Co. 1973.

17. Lillie. Class notes from EE 688, Computer Architecture. School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio. 1981.

18. Madnick, S. E. and J. J. Donovan. Operating Systems. New York, New York: Mc Graw-Hill Book Company, Inc., 1974.

19. Munro, J. Ian and Henra Sewanda. "Implicit Data Structures for Fast Search and Update," Journal of Computer and System Sciences, 21 (2). (October 1980).

20. Nakano, K. "Associatron- A Model of Associative Memory," IEEE Transactions on Systems, Man, and Cybernetics SMC-2: 380-388 (July 1972).

21. NASI 14700. User Guide: Relational Information Management (RIM). Seattle, Washington: Langley Research Center National Aeronautics and Space Administration, June 1981.

22. Nilsson, N. J. Learning Machines. New York, New York: McGraw-Hill Book Co. 1965.

23. Oppen, D. C. "Reasoning About Recursively Defined Data Structures." Journal of the ACM (27): 403-411 (July 1980).

24. Petersen, Steven V. CARTAM The Cartesian Access Method for Data Structures with N-Dimensional Keys. Headquarters, Strategic Air Command, Offutt Air Force Base, Nebraska, Reprint of material for his PhD dissertation at California Institute of Technology.

25. Rutledge. Class notes from EE 686, Information Structures. School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio. 1982.

26. ----- Class notes from EE 693, Software Engineering. School of Engineering, Air Force Institute of

Technology, Wright-Patterson AFB, Ohio. 1981.

27. Rux, P. T., F. W. Weingarten, and F. H. Young. Serial Associative Memories. Livermore, California: University of California, Lawrence Radiation Laboratory, December 13, 1966. (UCRL-70270).

28. Samet, H. "Region Representation: Quadtrees from Boundary Codes," Communications of the ACM: 163-170. (March 1980).

29. Seelandt, Karl G. "Computer Analysis and Recognition of Phoneme Sounds in Connected Speech." Unpublished MS thesis. School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio. December 1981.

30. Thurber, K. J., and L. D. Wald. "Associative and Parallel Processors," ACM Computing Surveys 7,(December 1975).

31. Weiderhold, G. Database Design. New York, New York: Mc Graw-Hill Book Company, Inc., 1974.

32. Wiener, N. Cybernetics: or Control and Communication in the Animal and Machine. Cambridge, Massachusetts: M. I. T. Press, 1961.

33. Winston, P. H. Artificial Intelligence. Reading, Massachusetts: Addison-Wesley Publishing Co. 1977.

34. Wirth, N. Algorithms + Data Structures = Programs. Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1976.

35. Woolridge, D. E. The Machinery of the Brain. New York, New York: McGraw-Hill Book Company, Inc., 1963.

APPENDIX A

MULTIDIMENSIONAL DATA STRUCTURES

A. Multidimensional Data Structures

Introduction

This appendix presents some of the main data structures which are currently used for multidimensional or associative data access. First some classical problems in data representation are discussed, then the recently acclaimed associative data structures are compared to the less well known CARTAM structure of Petersen's.

Petersen (Ref 24) has recently published a new method of representing multidimensional d-ta in sparsely populated regions. His method is based on an older popular method, but the older method has drawbacks which keep its performance below other more recent and radically different structures (Ref 6). Revising the older method, Petersen overcomes its major failings. The result is far superior to the other techniques for storing and manipulation multidimensional data.

Classic Data Structures

Numerical Mappings. One dimensional data is linear. If it has a smallest value and a finite number of larger values, these values can be mapped directly into consecutive computer memory addresses. As long as there are at least as many addresses as data points, data can be linearly stored in the computer memory. Two or higher dimension data must be mapped into a linear form to be stored in a computer memory. This is not difficult if the number of data points is small enough, but the space occupied grows exponentially with the number of dimensions mapped into the line. For example, with 10 data divisions in each dimension a three-dimensional space requires 1,000 memory addresses, while a four-dimensional space requires 10,000 memory addresses. The need for memory is explosive for higher dimension spaces. As long as the memory region is filled with useful data the numerical mapping is space efficient and gives rapid access.

Non-numerical Mappings. Most applications of data representation of high-dimensional data do not require every point of a grid to be represented. In fact, often the region can be less than ten percent filled. With this assumption, only a small portion of the addresses in memory would be used with the above defined mapping. To eliminate the waste of unused memory and allow efficient access of data without searching through unused data entries, logical data structures called trees, grids, and linked lists are used. The only data entries which are needed in these structures are those entries which exist in the represented region. Trees, grids, and linked lists are highly structured, and thus useful for only specific types of data access (Ref 15). Trees give rapid access, by value, to a single element. Grids give rapid access to all data entries "near-by" in value. Linked lists are used for data in linear organization.

Problems Encountered. Trees, grids, and linked lists (Ref 15) do not give flexible access . Each has its own use, but each also has its limitations. None can serve the tasks well supported by the others. To efficiently implement all of the relevant operations on data a more versatile structure is needed. Bentley (Ref 6) discusses a number of structures and the strong and weak points of each.

The data structures Bentley discusses (Ref 6) have substantial overhead in use of memory addresses and program code. Additional memory addresess are needed to store pointers to other regions within the data structure. This can add 10-20 percent overhead on storage, occasionally as much as 200-300 percent overhead. To traverse the structure, programs must be able to follow these pointers, thus the complexity of the programs is increased, and for certain awkward operations the time required to perform the operations is increased over the same operations on a linear representation (Ref 19).

Associative Data Structures

As currently applied, trees appear to be the most efficient structures to use for multidimensional or associative data access (Ref 6). Specialized tree forms have been created to allow multidimensional data access. The main forms are the quad tree, the K-D tree, and the data structure used in CARTAM.

Quad Trees. Many current sources on pattern recognition use a tree form called the quad tree to get the

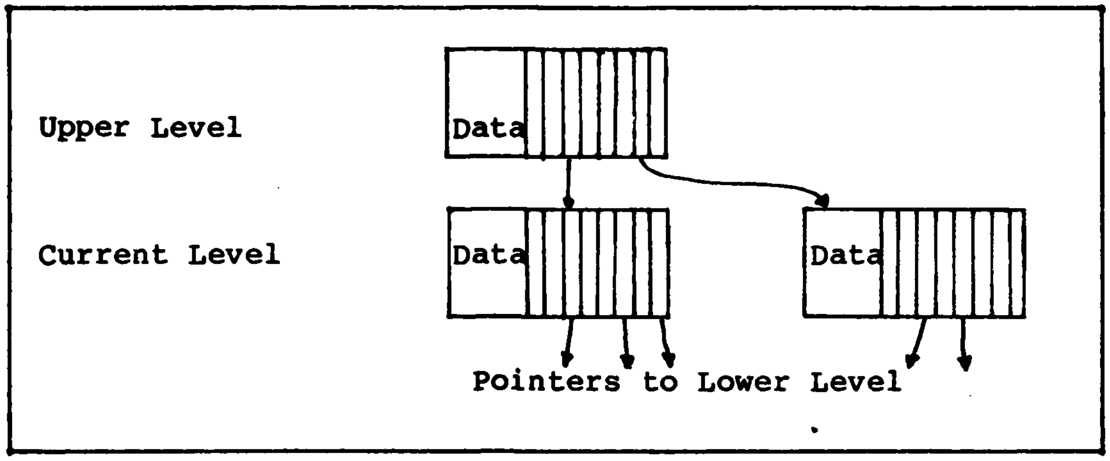


Fig 20. Quadtree Node
(3 Dimensions, 2 Sub-regions Occupied)

flexible access derived from trees yet retain a two dimensional search capability (Ref 11; 12; 19). The quad tree is actually an m-way tree with a fixed node size (Ref 10:496). However, this form is not space efficient when extended to more than two dimensions. The quad tree must have a pointer at each node for each half of each dimension. In two-dimensional space, a node divides its region into four quadrants, hence the name quad tree, and the node must have a pointer to each quadrant. If some of those quadrants are empty, the pointers must still exist to retain a standard node size and node format as in Figure 20. If the space were six-dimensional, there would have to be 64 pointers at each node, and the number of pointers doubles for each added dimension. The quadtree solves most of the problems of multidimensional access, but this wasted space, argues Bentley (Ref 6:407), makes the K-D tree better than the quad tree.

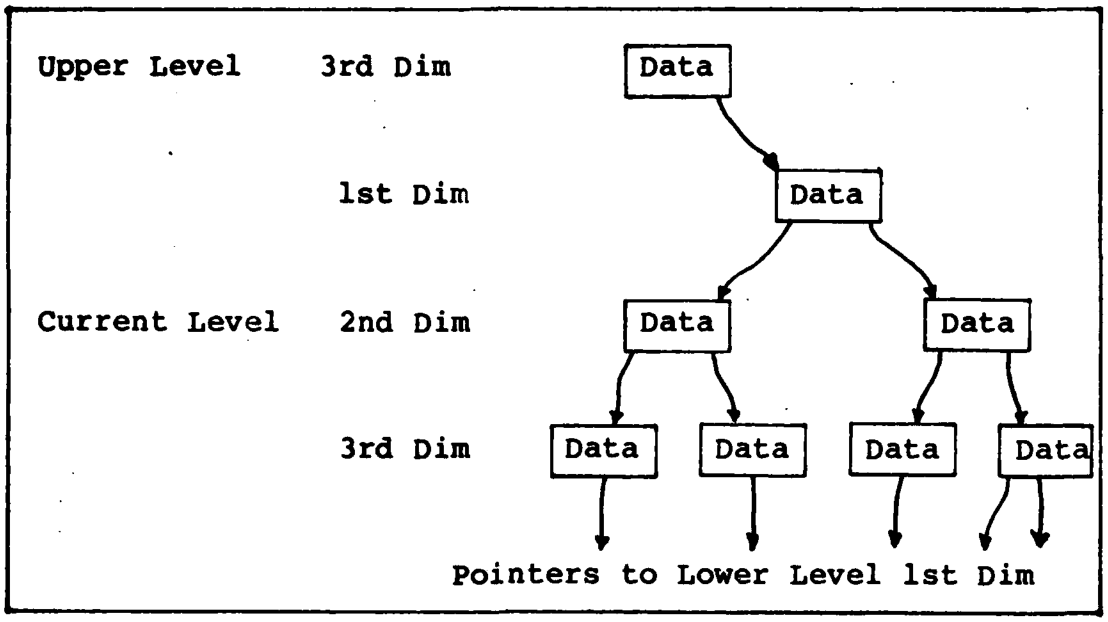


Fig 21. K-D Tree Levels
(3 Dimensions, 2 Sub-regions Occupied)

<u>K-D Trees</u>. The K-D tree is a binary tree where, for k dimensions, every k-th node represents a division in the same dimension at the next lower level in the structure. Each tree node represents a border between two ranges of data in a single dimension. In searching down a tree to access data, a decision is made at each node to reject half of the data, based on the value of a single dimension (see Figure 21), and continue the search in the other half of the data. This search is fine if every dimension is required for the search. If any dimension is not used in the search, then the choice to be made at the node representing that dimension is irrelevant to the problem, and thus both branches beneath the node must be searched. Bentley (Ref 6:404) defines these as K-D trees and claims they are the best trees for any search, showing comparisons with several

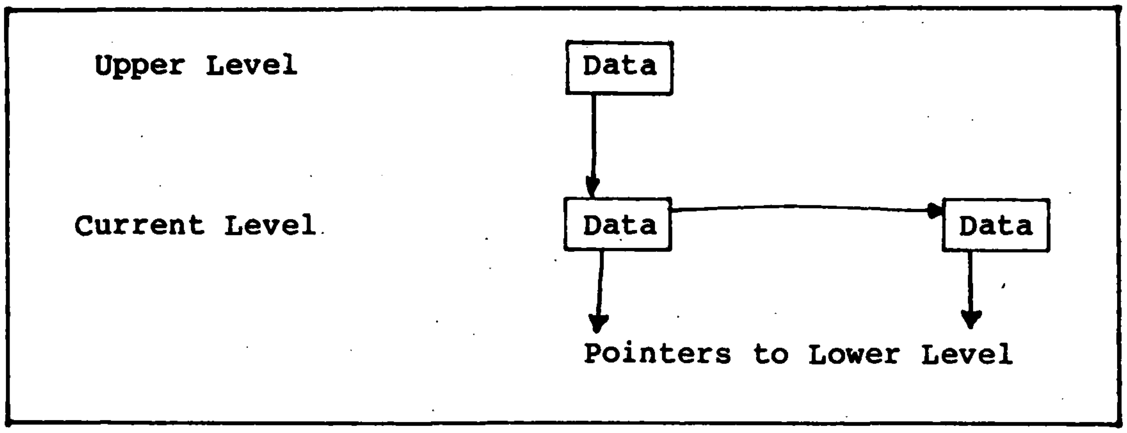


Fig 22. CARTAM Structure Levels
(3 Dimension, 2 Sub-regions Occupied)

other tree forms.

Another more serious problem with the K-D tree is that the decision criteria for every dimension except the first, at each level, must be duplicated for every cell at that level and dimension. This is not significant for small numbers of dimensions, but for six dimensions this means the sixth dimension's decision criteria may have to be copied 32 times.

CARTAM. Petersen (Ref 24:21-23) uses a version of the quadtree to represent multidimensional data. He creates a linked list of small cells which, together with their parent cell, make up the needed pointers of a single quadtree node as in Figure 22, and the data areas of the subordinate quadtree nodes. Being a linked list, only those cells which are needed to point to subordinate regions need be present, thus saving space where subordinate regions are not occupied.

The structure has the drawback of requiring a linear search of the linked list of subordinate cells to find those cells which satisfy the search criteria. For six dimensions this means only 64 elements possible per level, but for 10 dimensions this is 1024, doubling in size for each added dimension.

Petersen does not optimize his data storage, and leaves his structure with a larger data storage overhead than needed, however his revised node structure can be expanded to any number of dimensions.

Conclusions

Bentley (Ref 6:407) analyzed and rejected the quadtree, but it is still common in current literature (Ref 11; 12; 19). Petersen (Ref 24:21-23), by revising the quadtree slightly, eliminated the very difficulties Bentley had used to reject the quadtree. Petersen's structure, though requiring some optimization of data storage, is a more space efficient structure than the quad tree or K-D tree for representing multidimensional data in sparsely populated regions.

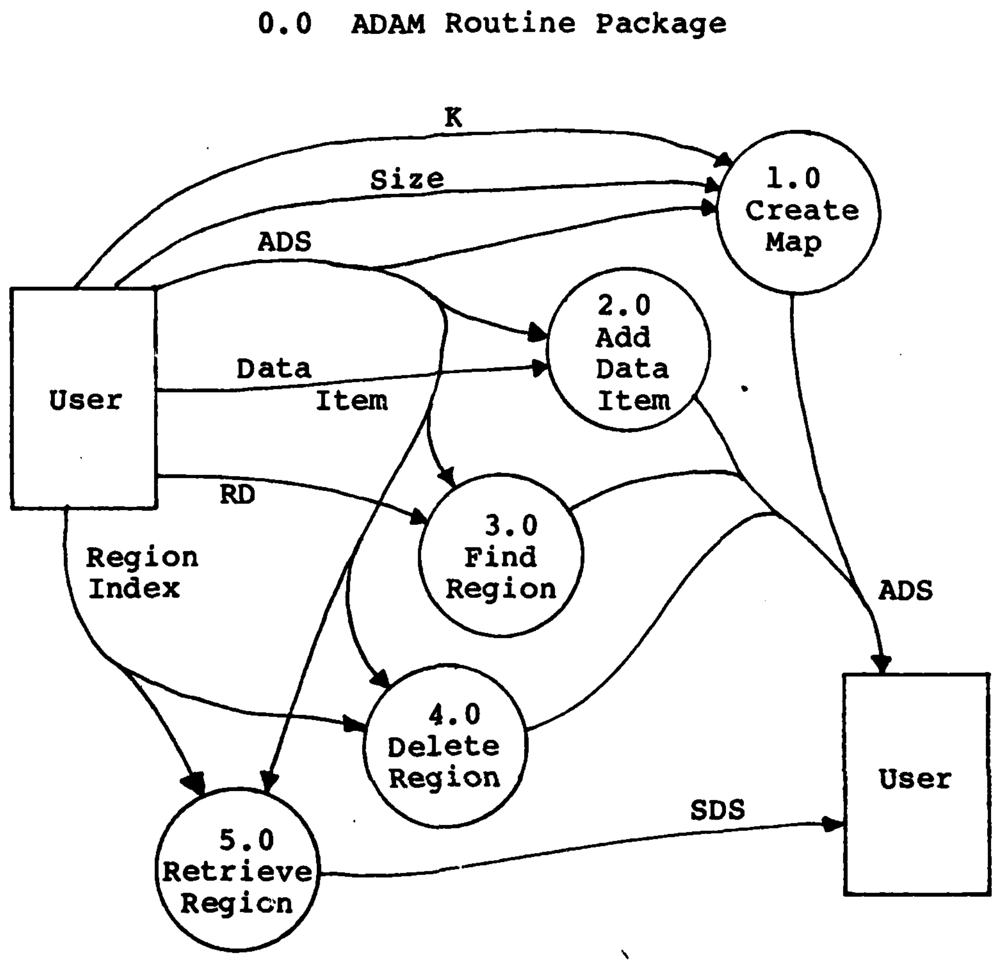
APPENDIX B

ADAM

DATA FLOW DIAGRAMS

(DFD)

Data Flow Diagram List



ADS--	Associative Data Set
SDS--	Sequential Data Set
RD---	Region Definition
Size-	Max Size of ADS
K----	Number of Dimensions

1.0	ACREATE
2.0	AADD
3.0	AFIND
4.0	ADELETE
5.0	ARETRIEVE

Fig 23. DFD of ADAM Algorithm Set

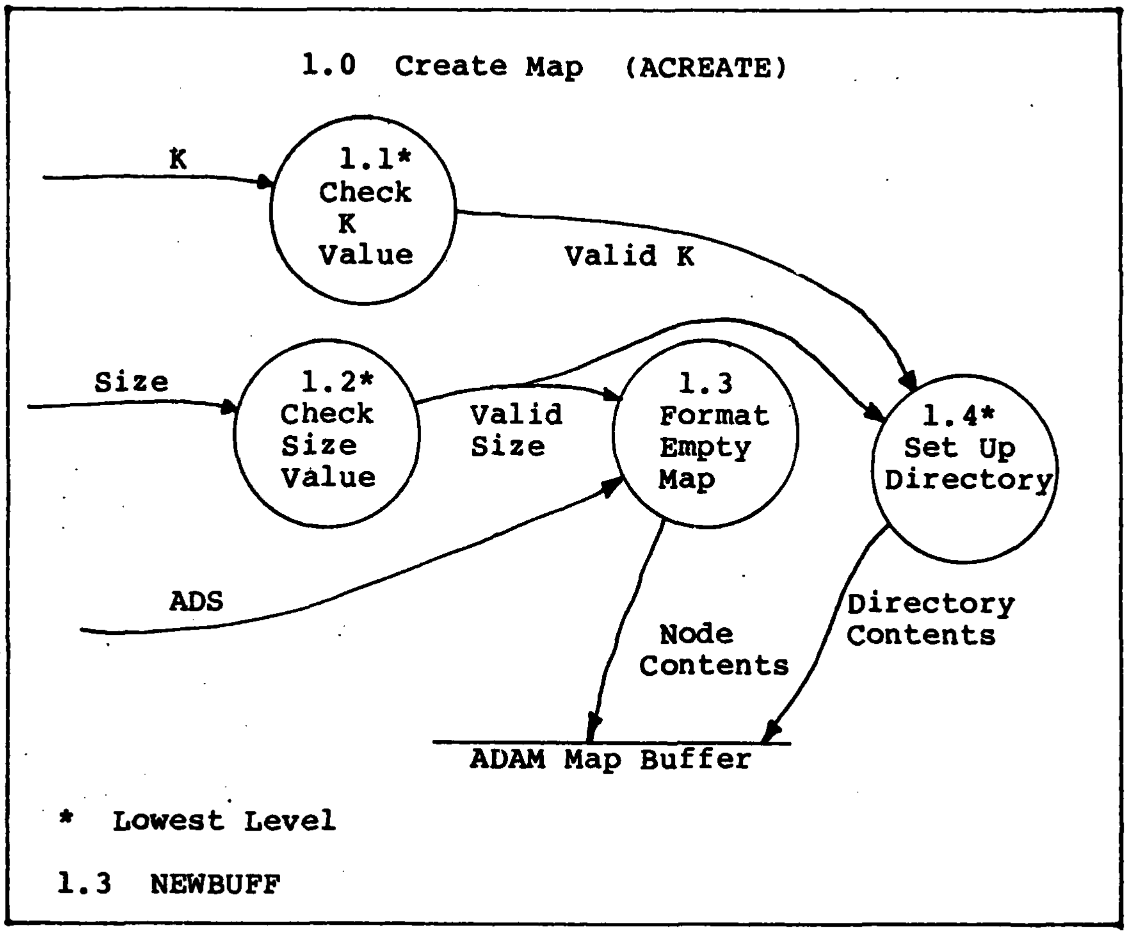


Fig 24. DFD of ACREATE

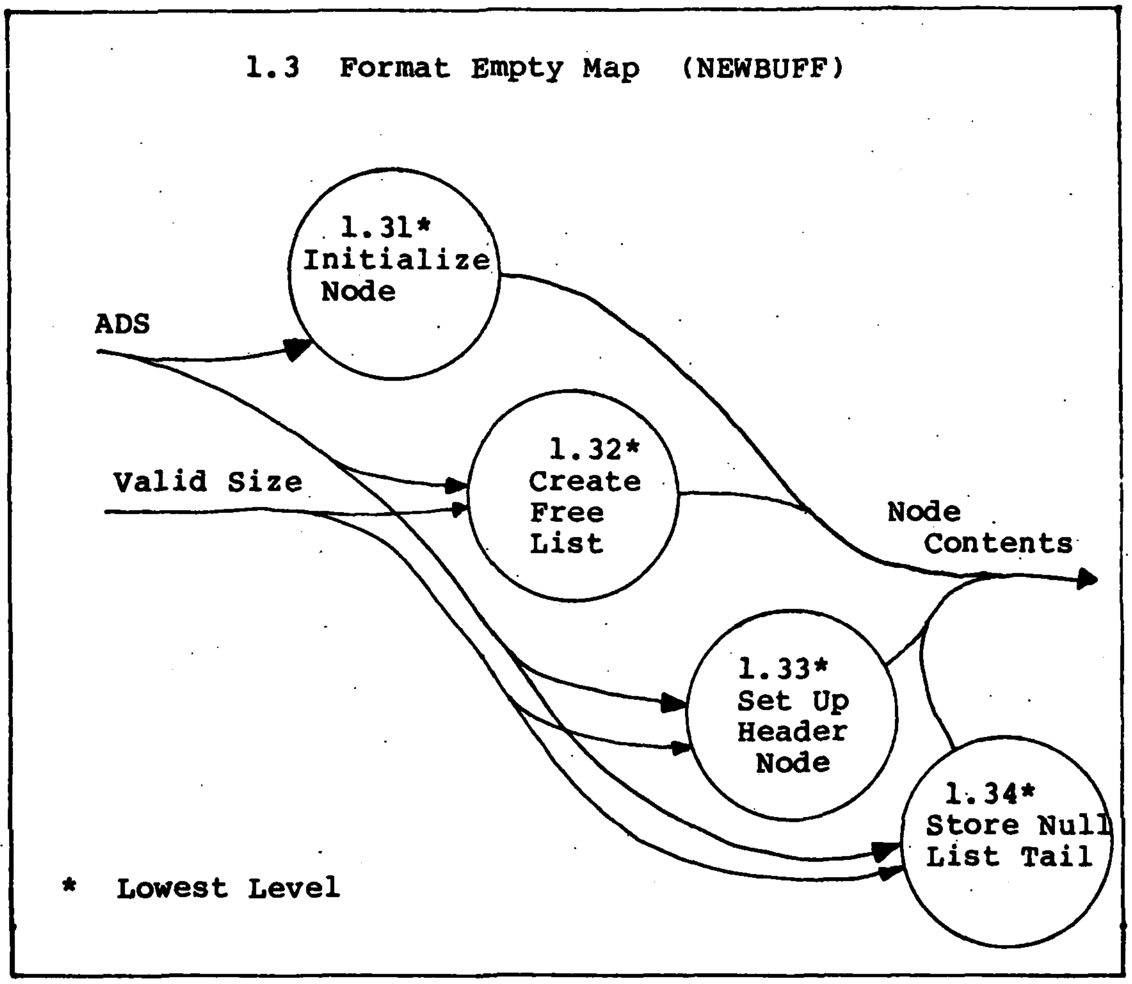


Fig 25. DFD of NEWBUFF

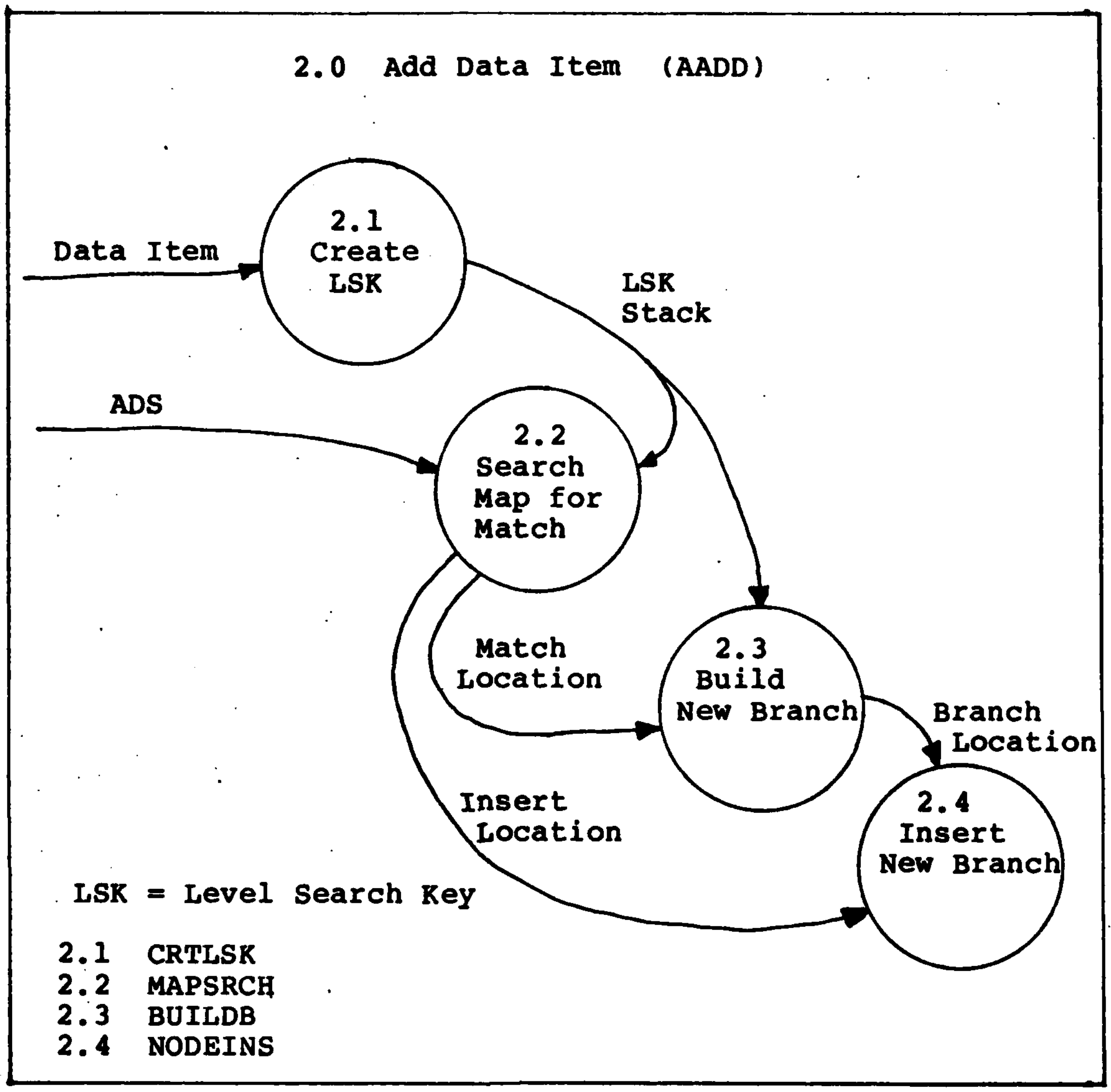
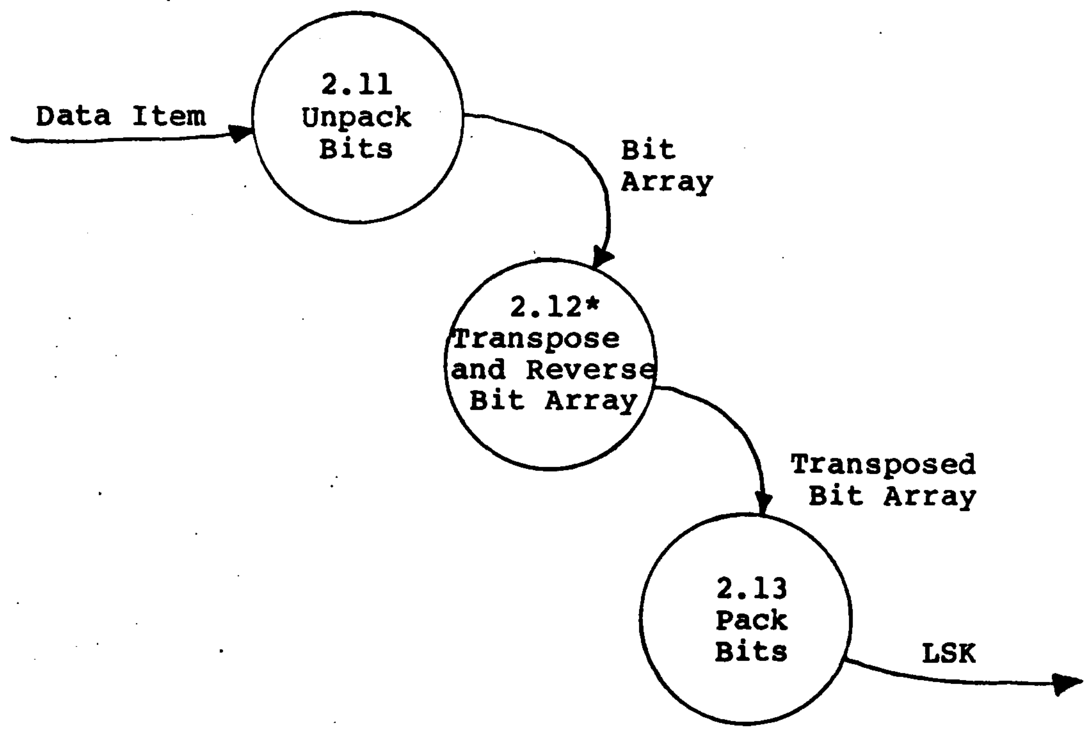


Fig 26. DFD of AADD

2.1 Create Level Search Keys (CRTLSK)



LSK = Level Search Key

* Lowest Level

2.11 BITPU
2.13 BITPU (see 2.11)

Fig 27. DFD of CRTLSK

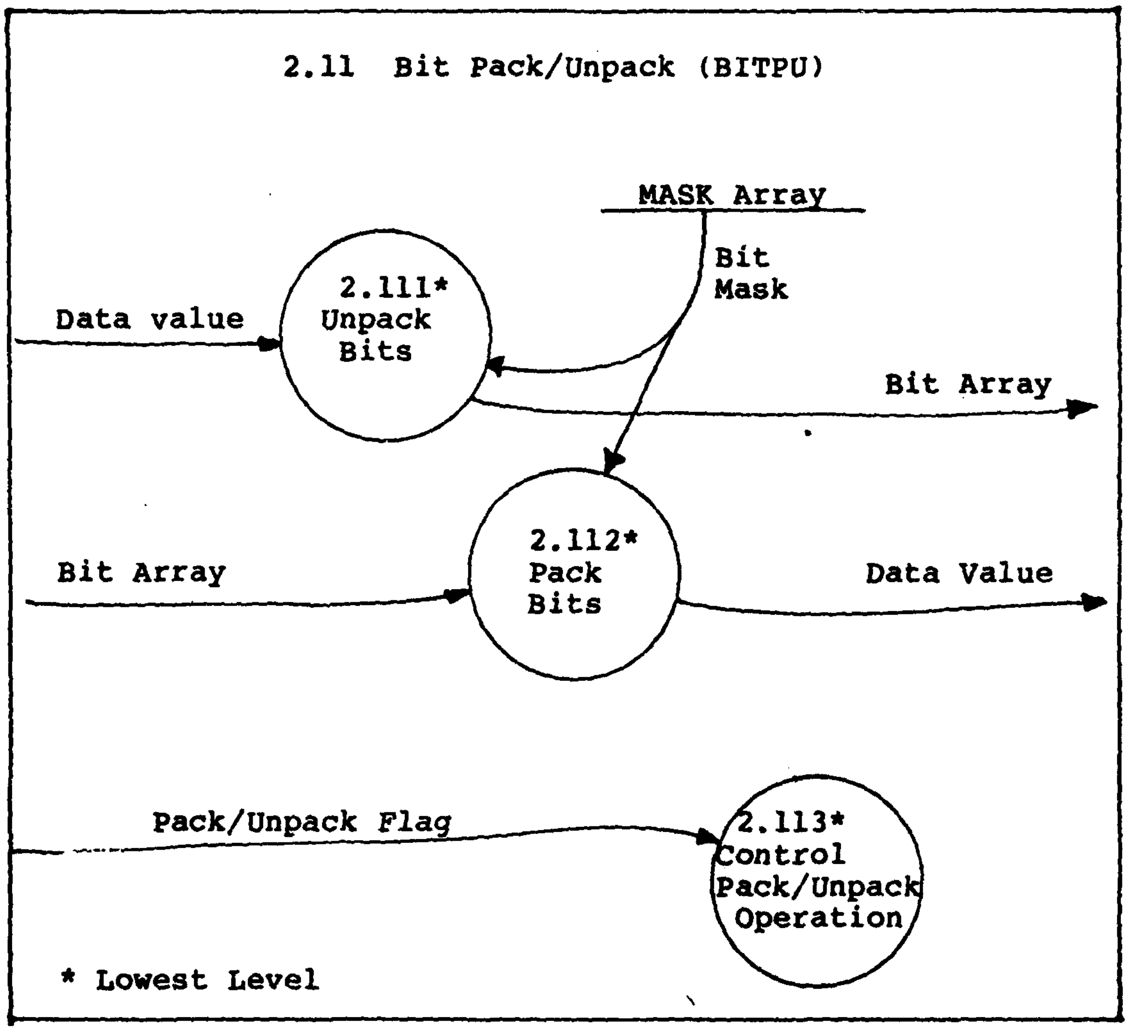


Fig 28. DFD of BITPU

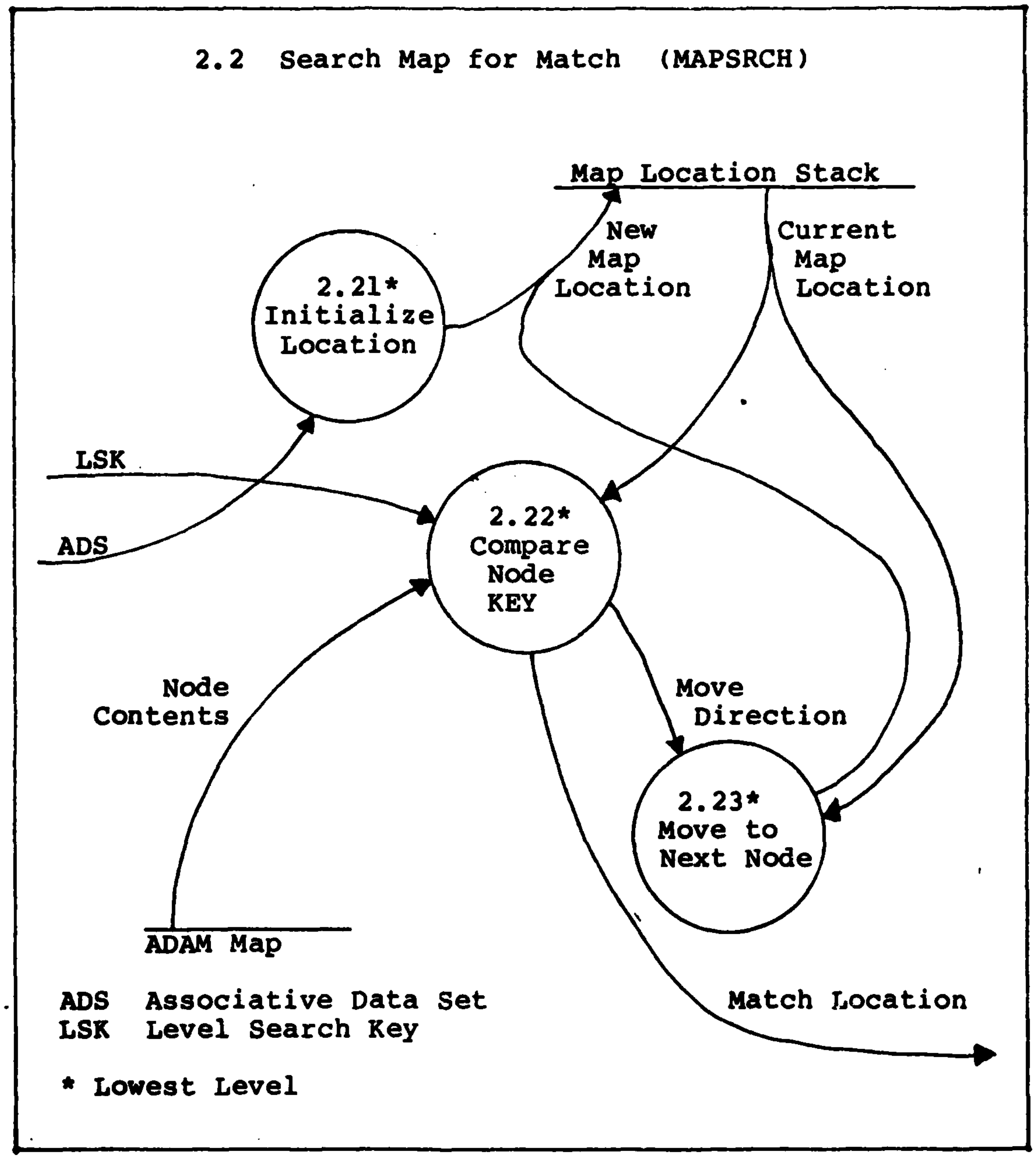


Fig 29. DFD of MAPSRCH

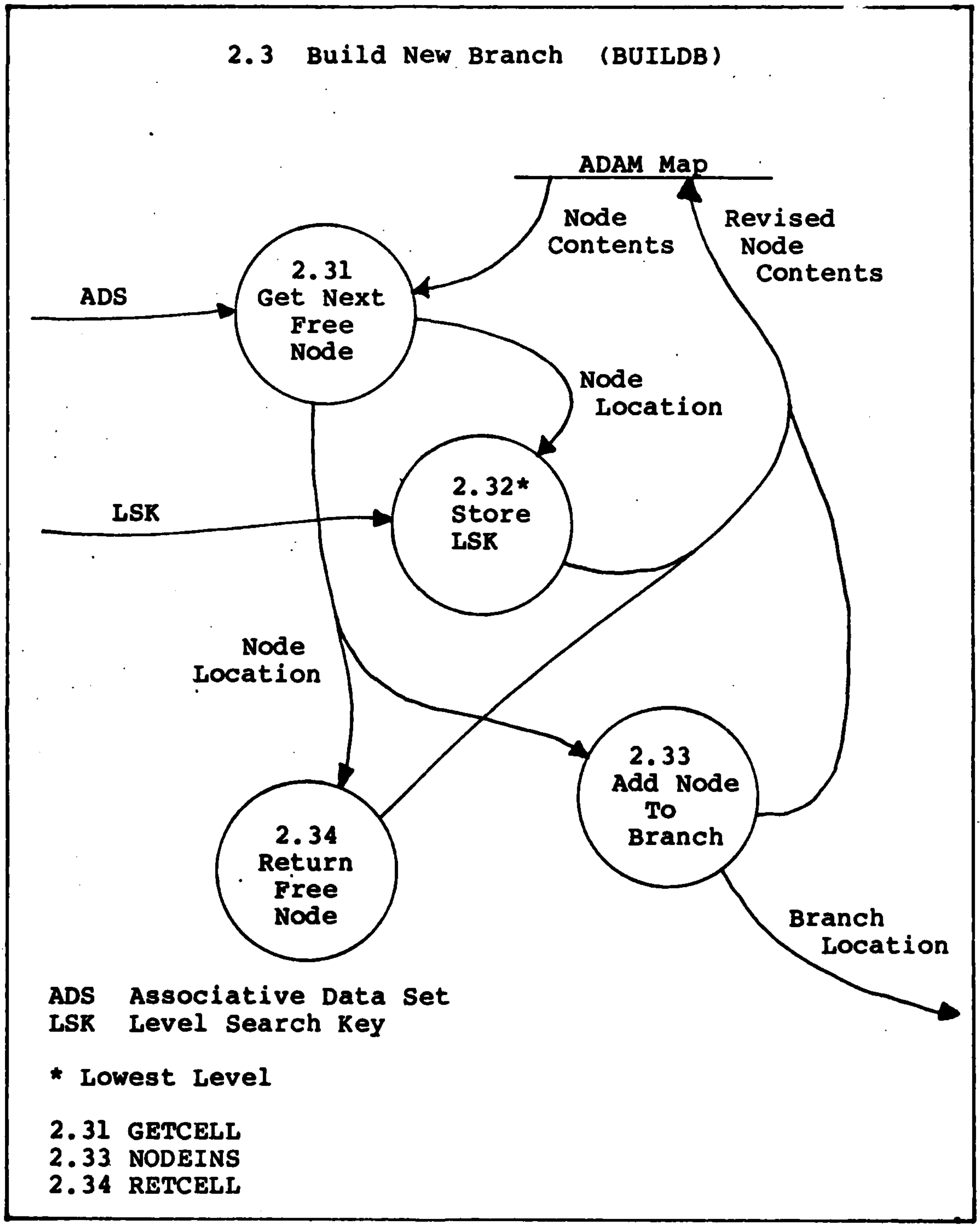


Fig 30. DFD of BUILDB

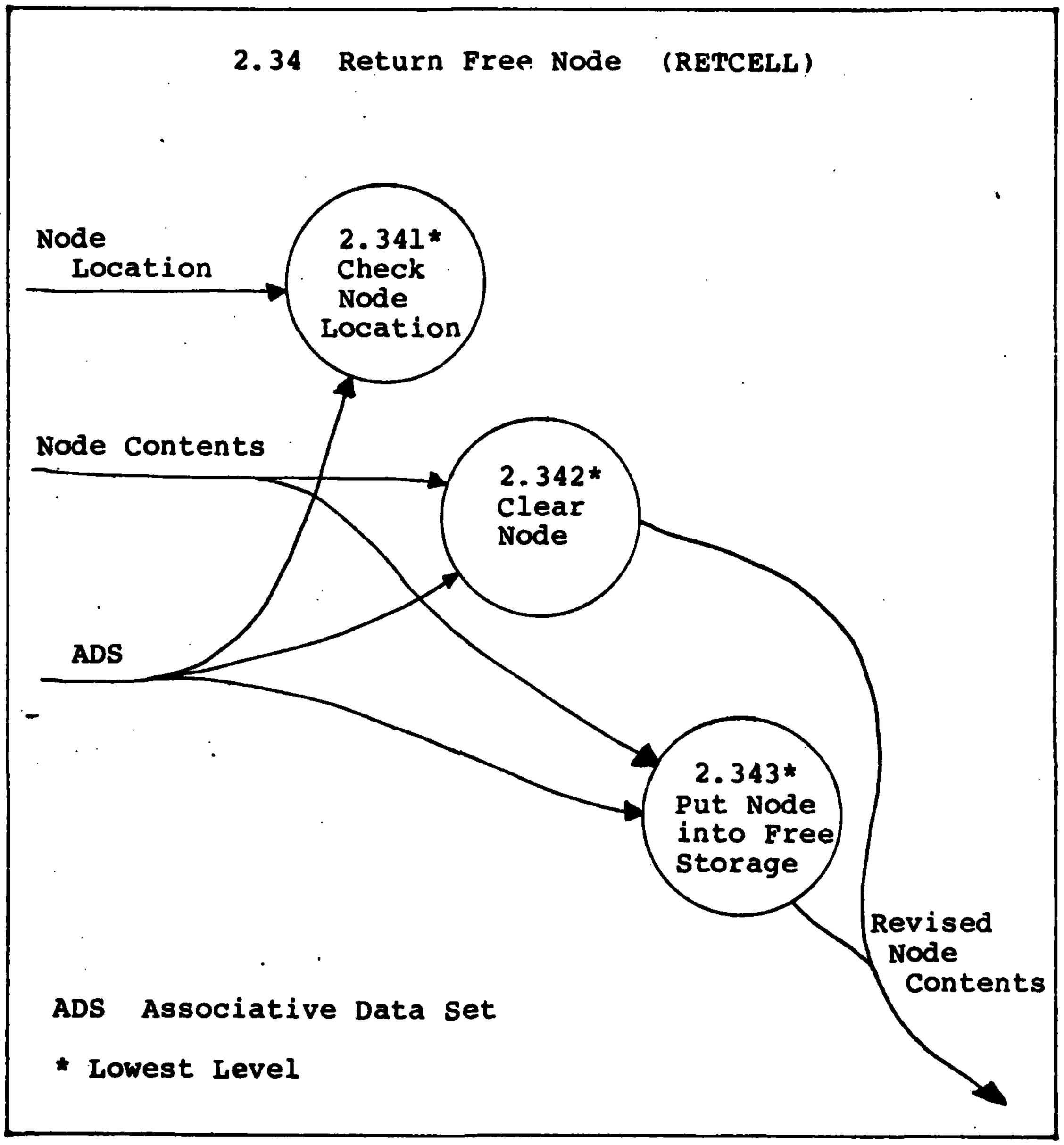


Fig 31. DFD of RETCELL

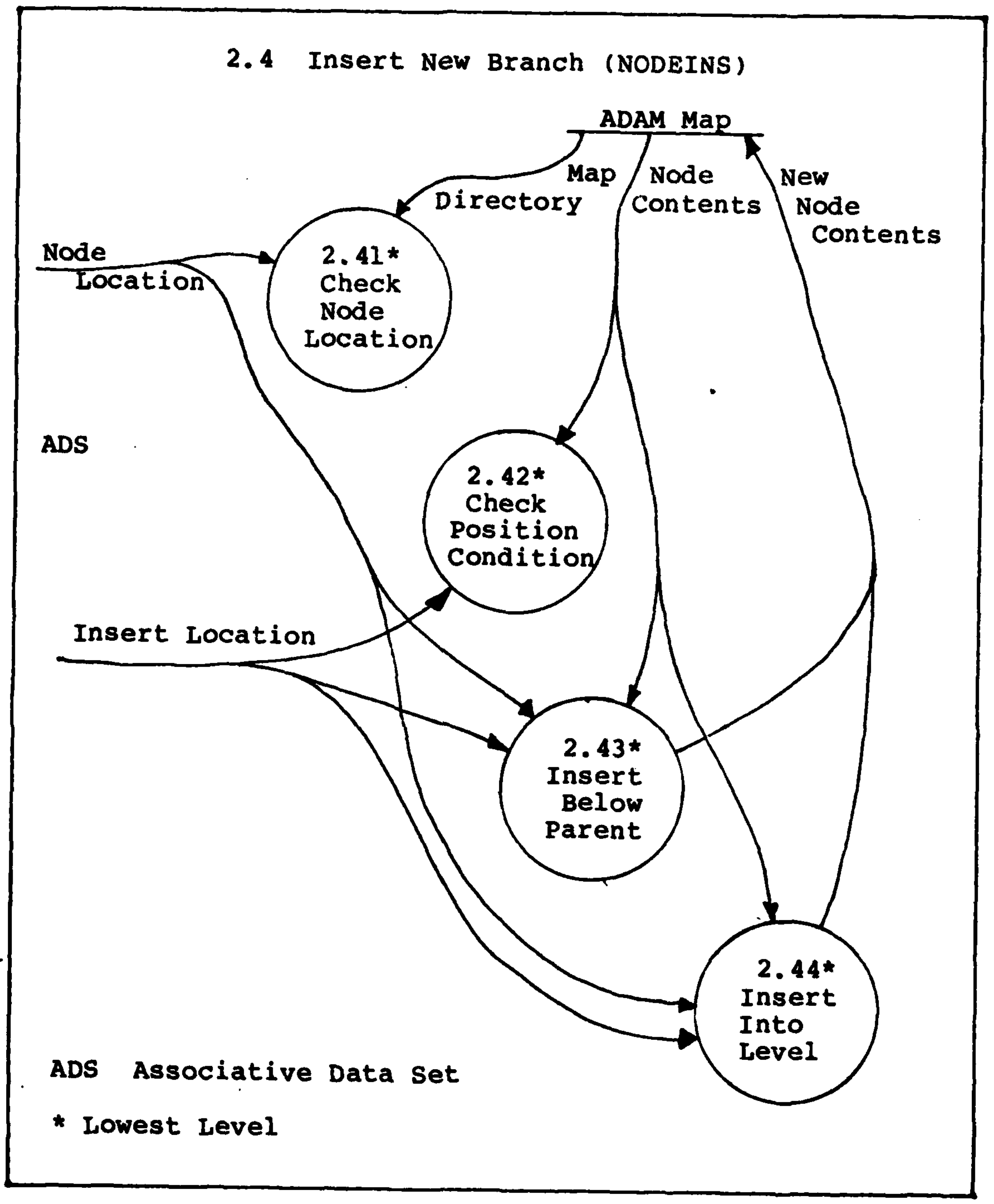


Fig 32. DFD of NODEINS

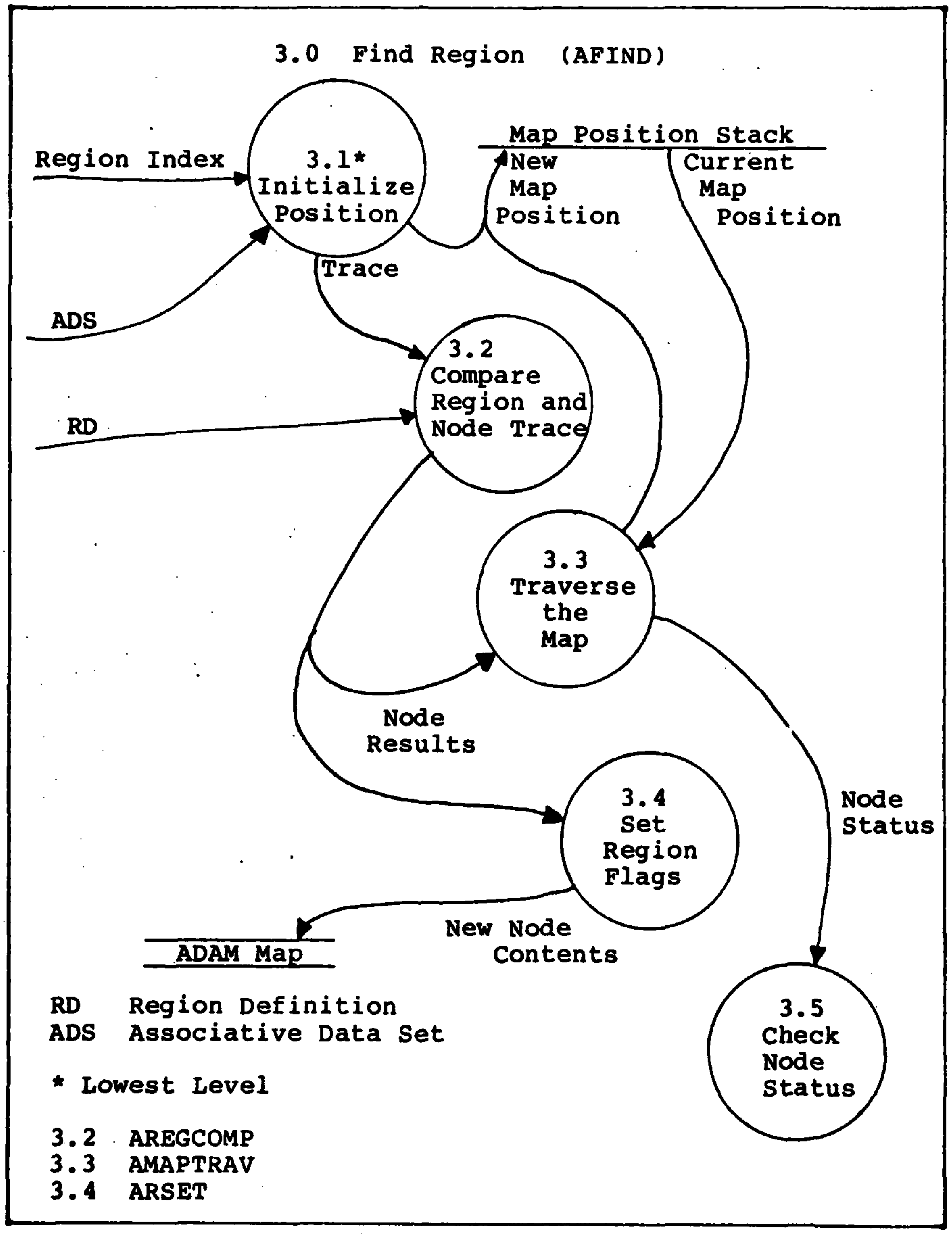


Fig 33. DFD of AFIND

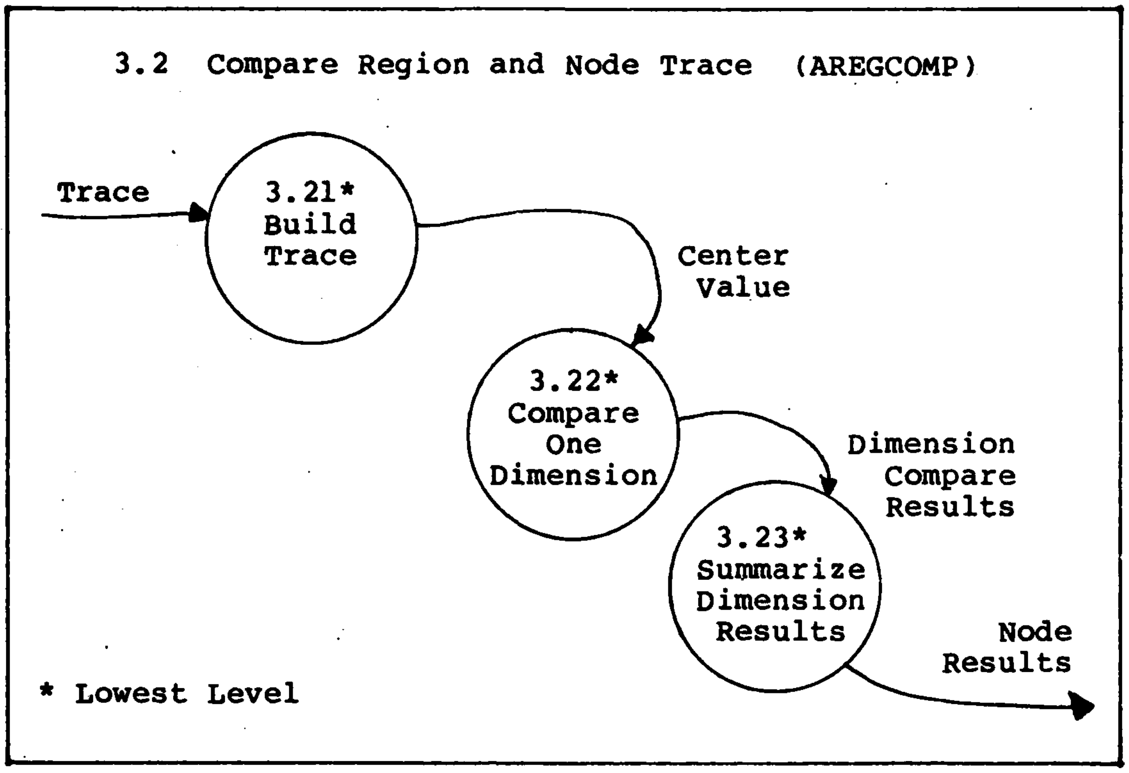

3.2 Compare Region and Node Trace (AREGCOMP)
Trace
3.21* Build Trace
Center Value
3.22* Compare One Dimension
Dimension Compare Results
3.23* Summarize Dimension Results
Node Results
* Lowest Level

Fig 34. DFD of AREGCOMP

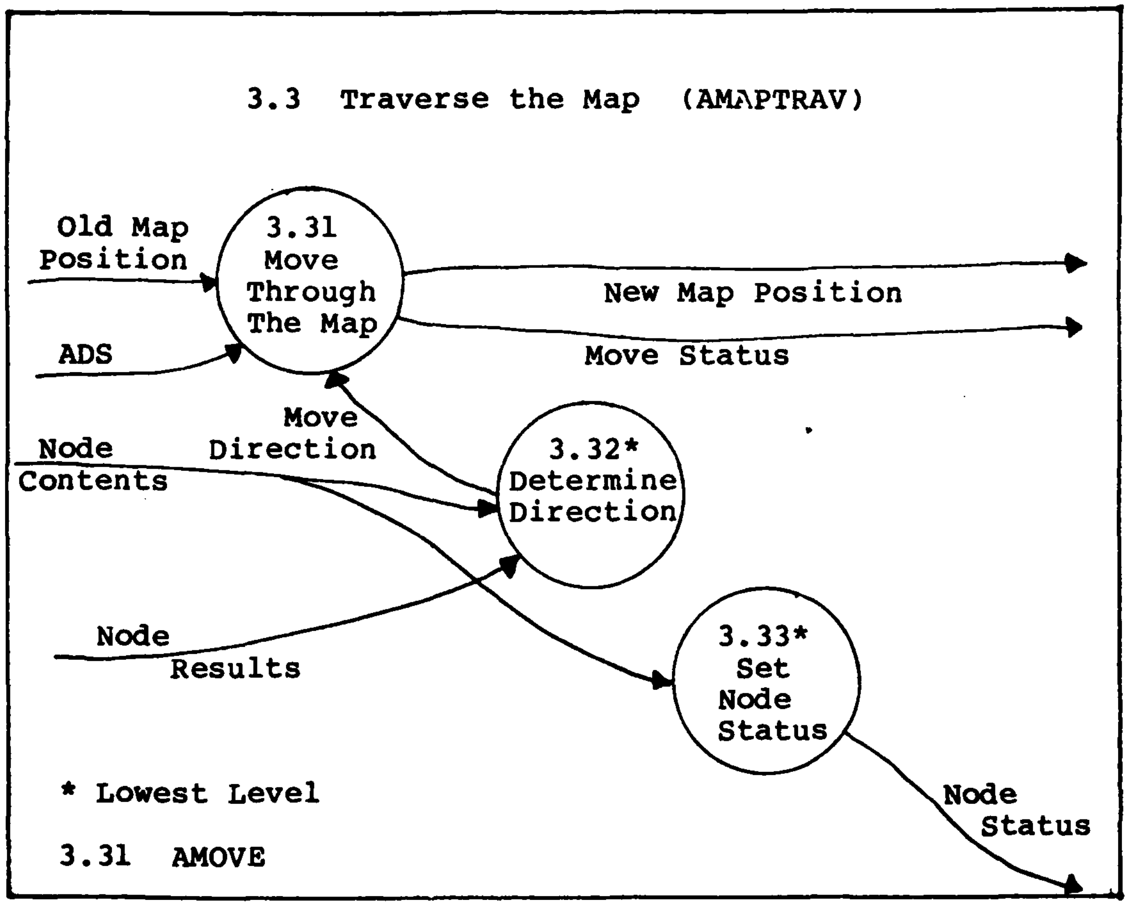


Fig 35. DFD of AMAPTRAV

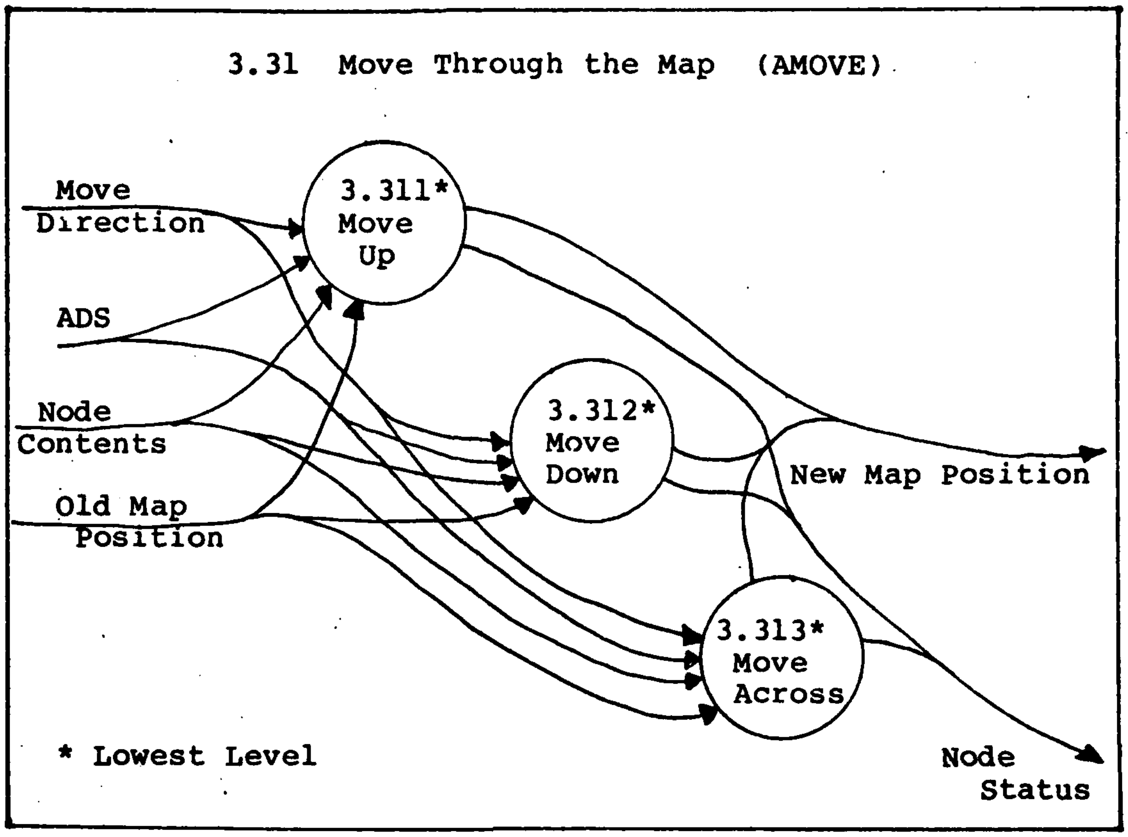


Fig 36. DFD of AMOVE

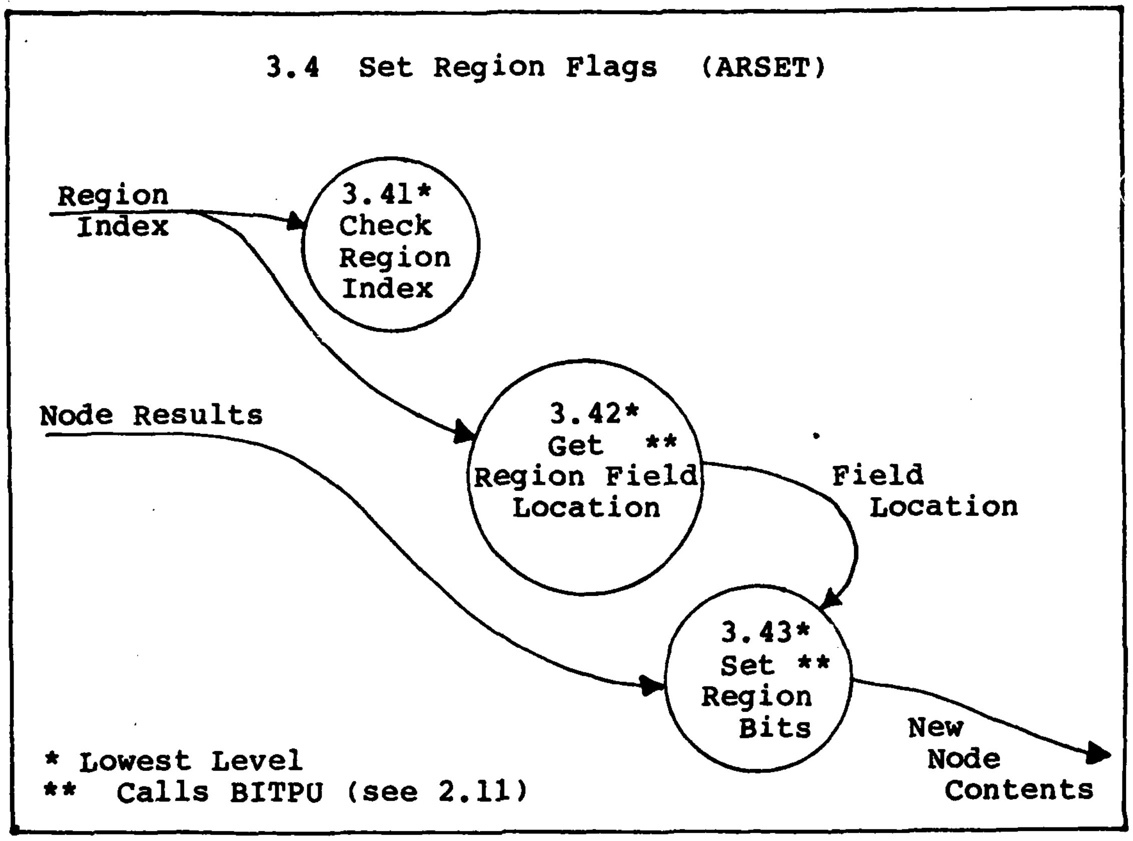
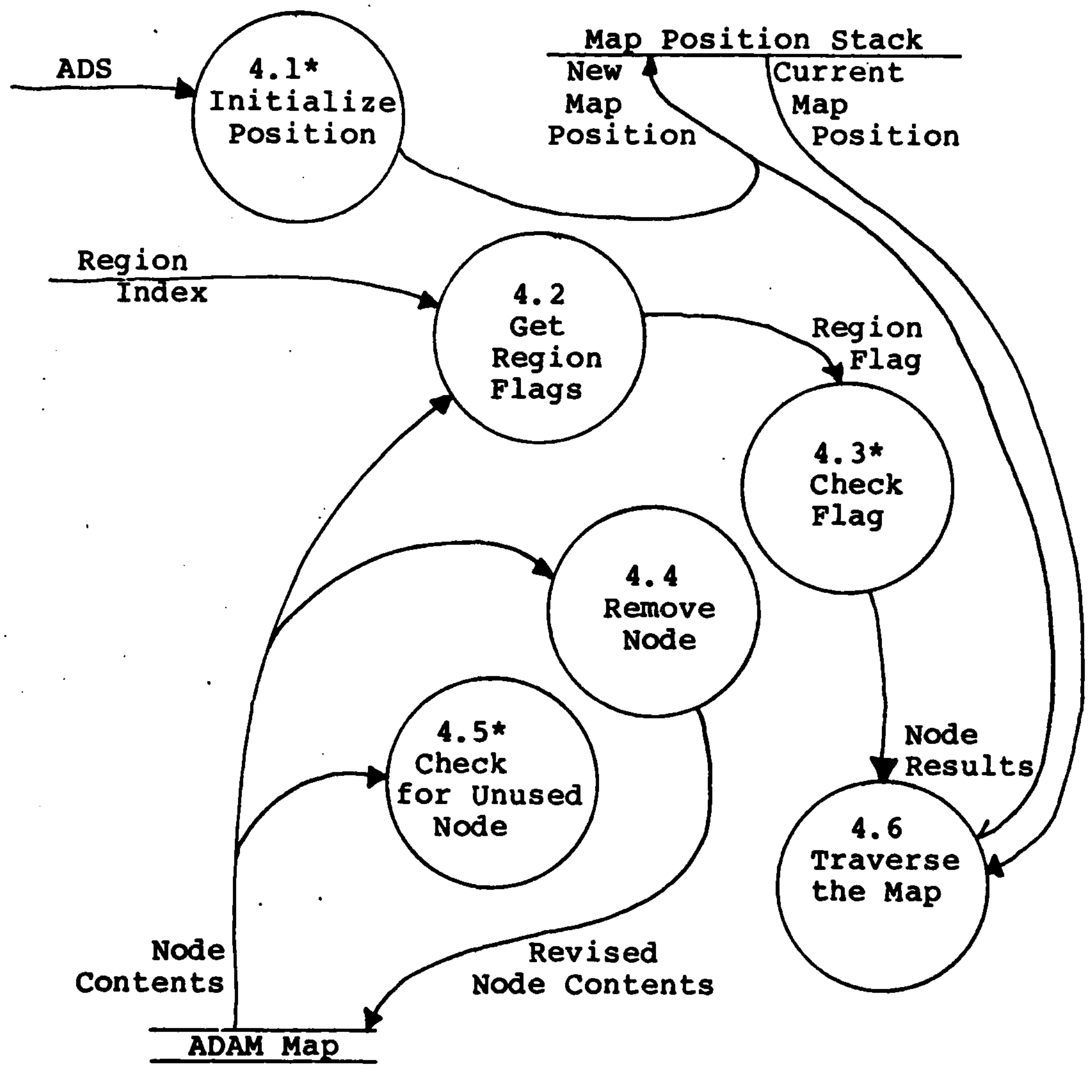


Fig 37. DFD of ARSET

4.0 Delete Region (ADELETE)



ADS Associative Data Set

* Lowest Level

4.2	ARSELECT
4.4	REMNODE
4.6	AMAPTRAV (see 3.3)

Fig 38. DFD of ADELETE

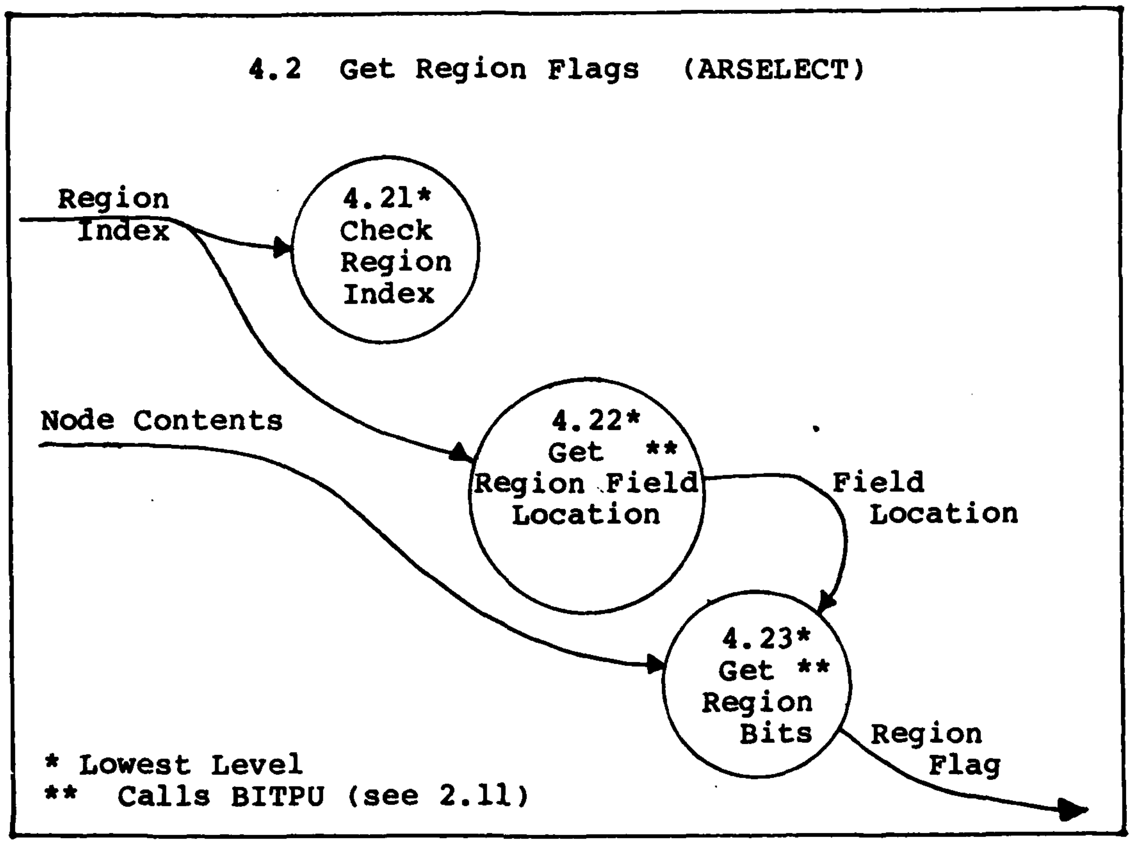


Fig 39. DFD of ARSELECT

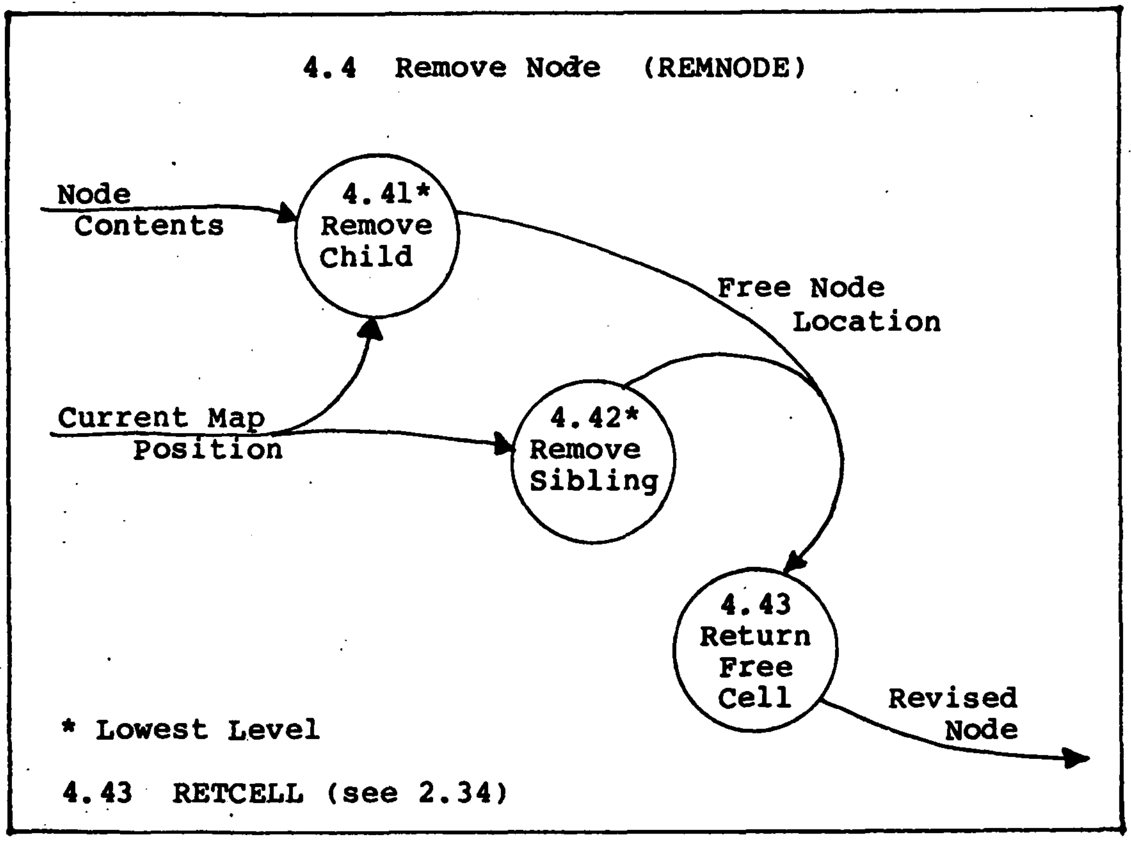
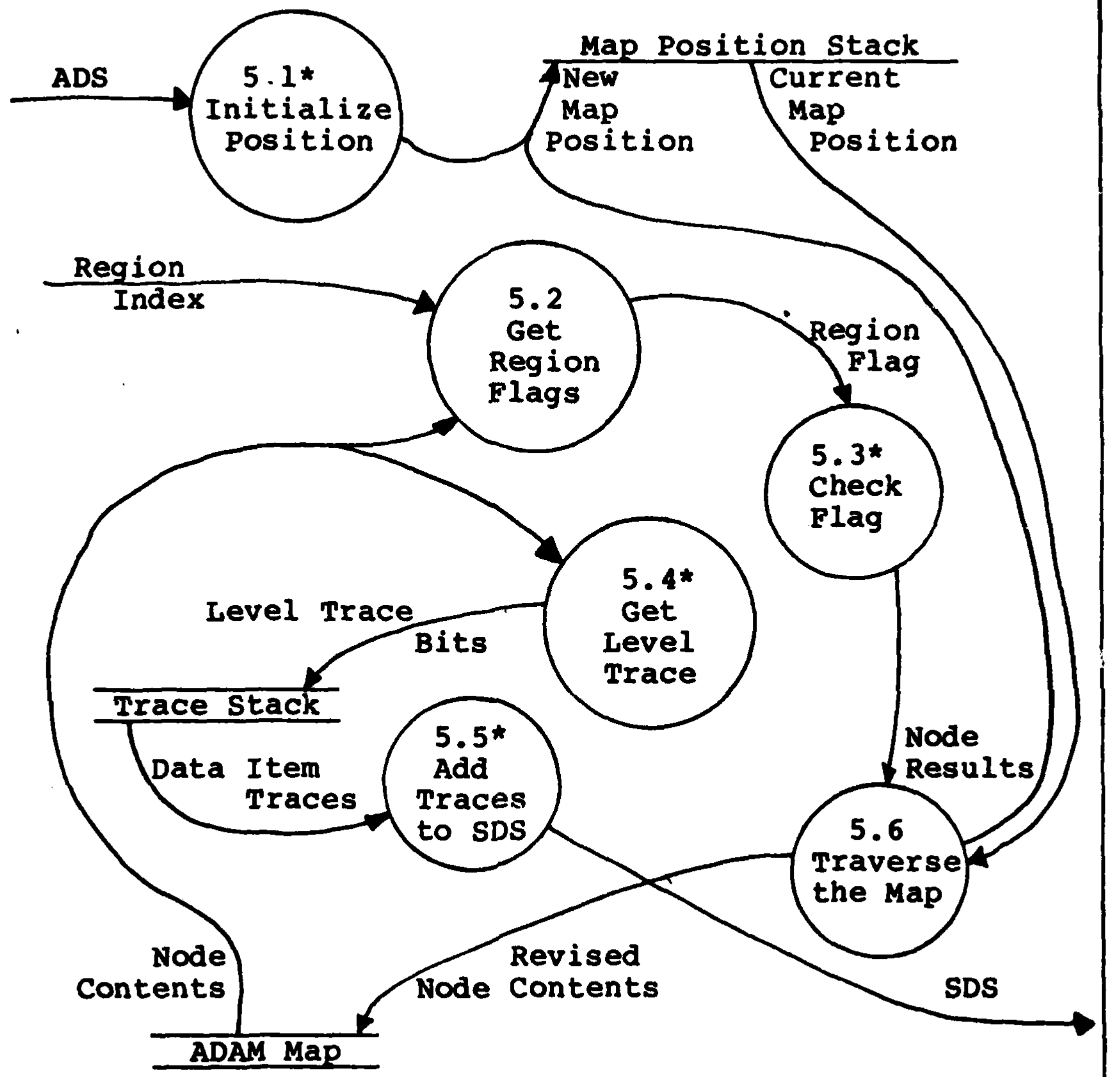


Fig 40. DFD of REMNODE

5.0 Retrieve Region (ARETRIEVE)



SDS	Sequential Data Set
ADS	Associative Data Set

* Lowest Level

5.2	ARSELECT
5.6	AMAPTRAV (see 3.3)

Fig 41. DFD of ARETRIEVE

APPENDIX C

DATA STRUCTURE DIAGRAMS

Data Structure Diagram List

Simple Variables Referenced

MAXNODES	-	Maximum ADAM map buffer size
MAXDIM	-	Maximum number of dimensions allowed
MAXREG	-	Maximum number of regions allowed
MAXDI	-	Maximum number of data items allowed in a sequential data set
NDIV	-	Number of data item vectors returned in a sequential data set
NUMDIM	-	Actual number of dimensions in a map
NUMLEV	-	Actual number of levels in a map

Associative Data Set (ADS)

ADAM Map
- Directory (1 time)
- Node (MAXNODES + 1 time)

Directory
- Number of Dimensions (I) (1 time)
- Number of Regions (I) (1 time)
- Number of Nodes ($\leq$ MAXNODES) (I) (1 time)
- Number of Levels (I) (1 time)

Node
- Sibling/Parent Pointer (I) (1 time)
- Child/Data Pointer (I) (1 time)
- Key Data Field (B) (MAXDIM times)
- Region Flag Field (2B) (MAXREG times)

(I) = integer variable

(B) = 1 bit field

(2B)= 2 bit field

Fig 42. ADAM Map Data Structure

Alternate View of ADS Buffer

$$\text{ADAM Buffer} \quad \begin{cases} \text{Directory (1 time)} \\ \text{Node (MAXNODES + 1 times)} \end{cases}$$

$$\text{Directory} \quad \left\{ \text{Integer word (4 times)} \right.$$

$$\text{Node} \quad \left\{ \text{Integer word (4 times)} \right.$$

Fig 43. ADAM Map Buffer Data Structure

Sequential Data Set (SDS)

$$\text{Sequential Data Set}\left\{\begin{array}{l}\text{Data Item Vector}\\ \qquad\text{(MAXDI times)}\end{array}\right.$$

$$\text{Data Item Vector}\left\{\begin{array}{l}\text{Integer (MAXDIM + 1 times)}\end{array}\right.$$

SDS Retrieval Data Structure

$$\text{SDS}\left\{\begin{array}{l}\text{Number of Data Item Vectors}\\ \qquad\text{(NDIV, I) (1 time)}\\ \text{Data Item Vector (NDIV times)}\end{array}\right.$$

$$\text{Data Item Vector}\left\{\begin{array}{l}\text{Data Item Value (I) (1 time)}\\ \text{Data Item Dimension Value (I)}\\ \qquad\text{(NUMDIM times)}\end{array}\right.$$

Fig 44. Sequential Retrieval Data Structure

Region Definition (RD) Data Set

RD	Low Value Array (1 time) High Value Array (1 time)
Low Value Array	Dimension Low Value (I) (NUMDIM times)
High Value Array	Dimension High Value (I) (NUMDIM times)

Fig 45. Region Definition Data Structure

Level Search Key (LSK) Data Set

$$\text{Level Search Key Vector}\ \left\{\begin{array}{l}\text{Level Search Key (I)}\\ \qquad\text{(NUMLEV times)}\end{array}\right.$$

Fig 46. Level Search Key Data Structure

Map Position Stack Data Set

$$\text{Map Position Stack}\ \left\{\begin{array}{l}\text{Position}\\ \qquad\text{(NUMLEV times)}\end{array}\right.$$

$$\text{Position}\ \left\{\begin{array}{l}\text{Parent Pointer (I) (1 time)}\\ \text{Current Pointer (I) (1 time)}\\ \text{Lastone Pointer (I) (1 time)}\\ \text{Nextone Pointer (I) (1 time)}\\ \text{Region Index (I) (1 time)}\end{array}\right.$$

Fig 47. Map Position Stack Data Structure

Trace Stack Data Set

$$\text{Trace Stack} \quad \left\{ \text{Level Search key (NUMLEV Times)} \right.$$

$$\text{Level Search Key} \quad \left\{ \begin{array}{l} \text{Dimension Decision Bit (B)} \\ \qquad \text{(NUMDIM times)} \end{array} \right.$$

Alternate View

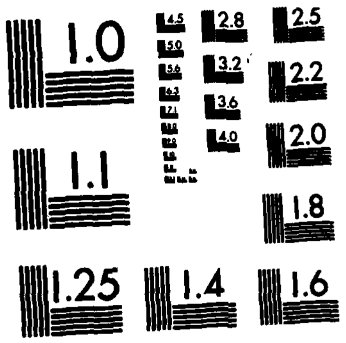
$$\text{Trace Stack} \quad \left\{ \begin{array}{l} \text{Dimension Trace (I)} \\ \qquad \text{(NUMDIM times)} \end{array} \right.$$

Fig 48. Trace Stack Data Structure

AD-A124 674   ASSOCIATIVE DATA ACCESS METHOD (ADAM)(U) AIR FORCE INST   3/3
OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING
J R HOLTEN DEC 82 AFIT/GCS/EE/82D-19

UNCLASSIFIED   F/G 9/2   NL

END
FILMED
DTIC


1.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
10
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

APPENDIX D

SOURCE LISTINGS

Listing Contents

( ) indicate references to DFD number in Appendix B

```
PROGRAM ADAMTEST;
     (* THIS PROGRAM TESTS THE ADAM PROCEDURES *)

CONST
     MAXNODES = 50;     (* SMALL BUFFER FOR NOW. *)
     MAXDIM = 10;       (* MAX NUMBER OF DIMENSIONS. *)
     MAXBIT = 16;       (* INTEGER LENGTH IN BITS. *)
          (* MAXBIT SHOULD ALWAYS BE BIGGER THAN MAXDIM. *)
     MAXDI = 100;        (* MAX DATA ITEMS IN A RETRIEVAL. *)
     MAXREG = 8;          (* MAX NUMBER OF FLAGGED REGIONS. *)

TYPE NODECELL = RECORD
          SPPTR,CDPTR:INTEGER;
          DATA:ARRAY (. 1..2 .) OF INTEGER;
          END;
     NODEARRAY = ARRAY (. 0..MAXNODES .) OF NODECELL;
     MAPDIRECT = RECORD
          NUMDIM,NUMREG,NUMNODES,NUMLEV:INTEGER;
          END;
     ADAMMAP = RECORD
          DIRECT:MAPDIRECT;
          NODE:NODEARRAY;
          END;
     PFILE = TEXT;
     BITSET = ARRAY (. 1..MAXBIT .) OF BOOLEAN;
     BITARRAY = ARRAY (. 1..MAXBIT .) OF BITSET;
     DIVEC = ARRAY (. 0..MAXBIT .) OF INTEGER;
     PUFLAG = (PACK,UNPACK);
     SCHRES = (NOTDONE,INSERT,MATCH);
     REGDEF = RECORD
          LOWVAL,HIGHVAL:DIVEC;
          END;
     SEQDS = ARRAY (. 0..MAXDI .) OF DIVEC;
     C8ARAY = ARRAY (. 1..8 .) OF CHAR;

VAR  MAP:ADAMMAP;
     MASK:ARRAY (. 1..MAXBIT .) OF INTEGER;
     PRFILE:PFILE;
     PRFLAG:(DISPLAY,PRINTER);
     BITFLG,VECFLG:BOOLEAN;         (* DEBUG FLAGS. *)

(*          THESE ARE THE INTERACTIVE SUPPORT ROUTINES.    *)

PROCEDURE INVAL(PROMPT:C8ARAY;VAR INDEX:INTEGER;
     MIN,MAX:INTEGER); EXTERNAL;
     (* USES PROMPT TO INPUT THE INDEX.  PROMPT RECURS UNTIL
     INPUT VALUE IS BETWEEN MIN AND MAX. *)

PROCEDURE INPDIV(PROMPT:C8ARAY;VAR DATVEC:DIVEC;
     NUMDIM:INTEGER); EXTERNAL;
     (* USES PROMPT TO INPUT THE DATA ITEM VECTOR DATVEC,
     WITH NUMDIM ENTRIES. *)

PROCEDURE OUTDIV(DATVEC:DIVEC;NUMDIM:INTEGER;VAR OFILE:PFILE);
```

```
     EXTERNAL;
     (* OUTPUTS THE DIVEC TO OFILE.  ASSUMES THERE ARE NUMDIM
     VALID ELEMENTS IN THE VECTOR. *)


(*              THESE ARE THE HIGHEST LEVEL PROCEDURES
                FOR MANIPULATING AN ADAM MAP.        *)

PROCEDURE AADD(VAR MAP:ADAMMAP;ITEM:DIVEC); EXTERNAL;
     (* ADDS DATA POINTS TO THE MAP.  USES INTERACTIVE DATA ENTRY. *)

PROCEDURE ACREATE(VAR MAP:ADAMMAP;SIZE,K,L:INTEGER); EXTERNAL;
     (* CREATES AN EMPTY ADAM MAP IN THE M-NODE MAP BUFFER
     AREA, SETTING UP THE DIRECTORY AND FREE STORAGE LIST. *)

PROCEDURE ADEBUG(VAR MAP:ADAMMAP); EXTERNAL;
     (* ALLOWS SOME DEBUGGING AIDS TO BE ACTIVATED/DEACTIVATED
     INTERACTIVELY, AND SOME DEBUGGING TOOLS TO BE USED. *)

PROCEDURE ADELETE(VAR MAP:ADAMMAP;INDEX:INTEGER); EXTERNAL;
     (* DELETES ALL THE DATA POINTS WITHIN THE SPECIFIED
     RETRIEVAL REGION. *)

PROCEDURE AFIND(VAR MAP:ADAMMAP;INDEX:INTEGER;RD:REGDEF);
     EXTERNAL;
     (* DEFINES A REGION WITHIN THE MAP FOR ALL RETRIEVED
     DATA POINTS. *)

PROCEDURE ARETRIEVE(VAR MAP:ADAMMAP;INDEX:INTEGER;SDS:SEQDS);
     EXTERNAL;
     (* TRANSLATES THE ADAM MAP DATA POINT ORGANIZATION INTO A
     SEQUENTIAL ORGANIZATION FOR SEQUENTIAL PROCESSING. *)
```

```
        (*          INTERACTIVE TOP LEVEL ROUTINES.        *)

PROCEDURE HELP;
     (* PRINTS OUT THE LEGAL COMMANDS AT THE TOP LEVEL
     AND GIVES THEIR FUNCTIONS. *)
BEGIN
WRITELN(' ADD  - - - - - - - ADD DATA TO THE MAP.');
WRITELN(' DEB  - - - - - - - DEBUG THE STRUCTURE.');
WRITELN(' DEL  - - - - - - - DELETE ALL DATA POINTS WITHIN A');
WRITELN('                    REGION.');
WRITELN(' FIND - - - - - - - FLAG THE NEEDED NODES TO DEFINE');
WRITELN('                    A REGION');
WRITELN(' GET  - - - - - - - RETRIEVE A REGION OF DATA POINTS');
WRITELN('                    INTO A SEQUENTIAL DATA FORM.');
WRITELN(' HELP - - - - - - - PRINT OUT HELP TABLE.');
WRITELN(' NEW  - - - - - - - CREATE A NEW MAP BUFFER.');
WRITELN(' STOP - - - - - - - STOP PROCESSING.');
END;   (* OF HELP *)
```

```
PROCEDURE ADAMCONTROL(VAR MAP:ADAMMAP);
     (* CONTROLS THE USE OF ADAM MAP FUNCTIONS INTERACTIVELY.
      *)

VAR  COMMAND:PACKED ARRAY (. 1..4 .) OF CHAR;
     RD:REGDEF;
     REGIND,K,SIZE:INTEGER;
     DATVEC:DIVEC;
     SDS:SEQDS;
     I,L:INTEGER;

BEGIN    (* ADAMCONTROL *)
HELP;
COMMAND:='    ';
WHILE COMMAND<>'STOP' DO
     BEGIN                 (* CYCLE UNTIL TOLD TO STOP. *)
     WRITE(' ADAM COMMAND= ');
     READLN(COMMAND);
     IF (COMMAND='ADD ') THEN
          BEGIN                      (* ADD A DATA POINT. *)
          INPDIV('DATA PT ',DATVEC,MAP.DIRECT.NUMDIM);
                                     (* INPUT IT. *)
          IF (DATVEC(.1.)>=0) THEN
               BEGIN                 (* GET THE POINT INDEX. *)
               INVAL('PT INDEX',DATVEC(.0.),1,MAXINT);
               AADD(MAP,DATVEC);              (* ADD TO MAP. *)
               END;
          END;
     IF (COMMAND='DEB ') THEN
          ADEBUG(MAP);        (* DEBUG A MAP. *)
     IF (COMMAND='DEL ') THEN
          BEGIN              (* DELETE A REGION. *)
          INVAL('REG IND ',REGIND,1,MAXREG);
          ADELETE(MAP,REGIND);
          END;
     IF (COMMAND='FIND') THEN
          BEGIN              (* DEFINE A REGION. *)
          INVAL('REG IND ',REGIND,1,MAXREG);
          INPDIV('REG MIN ',RD.LOWVAL,MAP.DIRECT.NUMDIM);
          IF (RD.LOWVAL(.1.))>=0 THEN
               BEGIN                 (* NO ABORT. *)
               INPDIV('REG MAX ',RD.HIGHVAL,MAP.DIRECT.NUMDIM);
               IF (RD.HIGHVAL(.1.)>=0) THEN
                    AFIND(MAP,REGIND,RD);
               END;
          END;
     IF (COMMAND='GET ') THEN
          BEGIN       (* GET ALL THE POINTS IN A REGION. *)
          INVAL('REG IND ',REGIND,0,MAXREG);
          ARETRIEVE(MAP,REGIND,SDS);
          IF(SDS(.0,0.)<=0) THEN
               WRITELN(PRFILE,
                    ' NO DATA POINTS IN THE REGION.')
          ELSE BEGIN
```

```
                    WRITELN(PRFILE,' RETRIEVED DATA POINTS.');
                    K:=MAP.DIRECT.NUMDIM;
                    I:=1;
                    WHILE (I<=SDS(.0,0.)) DO
                         BEGIN          (* WRITE OUT THE VECTORS. *)
                         WRITE(PRFILE,SDS(.I,K+1.),'  ');
                         OUTDIV(SDS(.I.),K,PRFILE);
                         I:=I+1;
                         END;
                    WRITELN(PRFILE);
                    END;
               END;
          IF (COMMAND='HELP') THEN
               HELP;        (* PRINT HELP INFORMATION. *)
          IF (COMMAND='NEW ') THEN
               BEGIN
               INVAL('BUF SIZE',SIZE,50,MAXNODES);
               INVAL('NUM DIM ',K,1,MAXDIM);
               INVAL('NUM LEV ',L,3,MAXBIT);
               ACREATE(MAP,SIZE,K,L);     (* FORMAT NEW MAP.*)
               END;
          IF (COMMAND='STOP') THEN WRITELN(' STOP REQUESTED.');
          END;
     END;       (* OF ADAMCONTROL *)
```

```
PROCEDURE INITIALIZE;
     (* INITIALIZES SYSTEM VARIABLES. *)
VAR  I,MVAL:INTEGER;
BEGIN
MVAL:=1;
I:=1;
WHILE (I<MAXBIT) DO
     BEGIN          (* SET BIT MASKS. *)
     MASK(. I .):=MVAL;
     I:=I+1;
     IF (I<MAXBIT) THEN   (* PREVENT OVERFLOW. *)
          MVAL:=MVAL*2;
     END;
PRFLAG:=DISPLAY;
BITFLG:=FALSE;
VECFLG:=FALSE;
END;  (* OF INITIALIZE *)
```

```
BEGIN             (* ADAMTEST *)
REWRITE(PRFILE);
INITIALIZE;
WRITELN(' ADAM TEST ROUTINE. ');
ADAMCONTROL(MAP);
WRITELN(' END OF ADAM TEST.' );
END.
```

```
PROGRAM ADAMIO;
     (* THIS PROGRAM TESTS THE ADAM PROCEDURES *)

CONST
     MAXNODES = 50;     (* SMALL BUFFER FOR NOW. *)
     MAXDIM = 10;       (* MAX NUMBER OF DIMENSIONS. *)
     MAXBIT = 16;       (* INTEGER LENGTH IN BITS. *)
          (* MAXBIT SHOULD ALWAYS BE BIGGER THAN MAXDIM. *)
     MAXDI = 100;        (* MAX DATA ITEMS IN A RETRIEVAL. *)

TYPE NODECELL = RECORD
          SPPTR,CDPTR:INTEGER;
          DATA:ARRAY (. 1..2 .) OF INTEGER;
          END;
     NODEARRAY = ARRAY (. 0..MAXNODES .) OF NODECELL;
     MAPDIRECT = RECORD
          NUMDIM,NUMREG,NUMNODES,NUMLEV:INTEGER;
          END;
     ADAMMAP = RECORD
          DIRECT:MAPDIRECT;
          NODE:NODEARRAY;
          END;
     PFILE = TEXT;
     BITSET = ARRAY (. 1..MAXBIT .) OF BOOLEAN;
     BITARRAY = ARRAY (. 1..MAXBIT .) OF BITSET;
     DIVEC = ARRAY (. 0..MAXBIT .) OF INTEGER;
     PUFLAG = (PACK,UNPACK);
     SCHRES = (NOTDONE,INSERT,MATCH);
     REGDEF = RECORD
          LOWVAL,HIGHVAL:DIVEC;
          END;
     SEQDS = ARRAY (. 0..MAXDI .) OF DIVEC;
     C8ARAY = ARRAY (. 1..8 .) OF CHAR;

VAR  MAP:ADAMMAP;
     MASK:ARRAY (. 1..MAXBIT .) OF INTEGER;
     PRFILE:PFILE;
     PRFLAG:(DISPLAY,PRINTER);
     BITFLG,VECFLG:BOOLEAN;         (* DEBUG FLAGS. *)
```

```
   (*   I/O ROUTINES.   *)

PROCEDURE OUTBITS(OFILE:PFILE;BITS:BITSET;NUM:INTEGER);
     (* OUTPUTS NUM BITS TO OFILE. *)
VAR  I:INTEGER;
BEGIN
I:=1;
WHILE(I<=NUM) DO
     BEGIN      (* OUTPUT A BIT. *)
     WRITE(OFILE,' ',BITS(.I.):1);
     I:=I+1;
     END;
WRITELN(OFILE);
END;   (* OF OUTBITS *)
```

```
PROCEDURE OUTVEC(VAR VEC:DIVEC);
     (* OUTPUTS ALL OF VEC FOR DEBUGGI[illegible]  *)
VAR  I:INTEGER;
BEGIN
I:=0;
WHILE (I<=MAXBIT) DO
     BEGIN                (* OUTPUT IN HEX. *)
     IF ((I+1)MOD 9 = 0) THEN WRITELN(PRFILE);
     WRITE(PRFILE,'     ',VEC(.I.):4 HEX);
     I:=I+1;
     END;
WRITELN(PRFILE);
I:=0;
WHILE (I<=MAXBIT) DO
     BEGIN                (* OUTPUT IN INTEGER. *)
     IF ((I+1) MOD 9 = 0) THEN WRITELN(PRFILE);
     WRITE(PRFILE,'  ',VEC(.I.):6);
     I:=I+1;
     END;
WRITELN(PRFILE);
END;   (* OF OUTVEC *)
```

```
          (* INTERACTIVE I/O ROUTINES. *)

PROCEDURE INVAL(PROMPT:C8ARAY;VAR VALUE:INTEGER;
     MIN,MAX:INTEGER);
     (* USES PROMPT TO INPUT A VALUE FROM THE USER.
     IGNORES ALL VALUES OUTSIDE THE INTERVAL MIN TO MAX. *)
BEGIN
VALUE:=MIN-1;
WHILE (VALUE<MIN) OR (MAX<VALUE) DO
     BEGIN                        (* LOOP UNTIL VALID. *)
     WRITE(' INPUT ',PROMPT,' >');
     READLN(VALUE);
     END;
END;
```

```
PROCEDURE OUTDIV(DATVEC:DIVEC;K:INTEGER;VAR OFILE:PFILE);
     (* OUTPUTS K ELEMENTS OF DATVEC TO PFILE. *)
VAR  I:INTEGER;
     UNSCALE,RVALUE:REAL;
BEGIN
RVALUE:=MAXINT+1.0;
UNSCALE:=1.0/RVALUE;
I:=1;
WHILE(I<=K) DO
     BEGIN
     IF (I MOD 5 = 0) THEN WRITELN(OFILE);
     RVALUE:=DATVEC(.I.)*UNSCALE;
     WRITE(OFILE,'  ',RVALUE:7:5);
     I:=I+1;
     END;
WRITELN(OFILE);
END;
```

```
PROCEDURE INPDIV(PROMPT:C8ARAY;VAR DATVEC:DIVEC;K:INTEGER);
     (* INTERACTIVELY INPUTS THE DATA ITEM VECTOR. *)
VAR  INDEX,IDVAL:INTEGER;
     DATAVALUE,RMAXINT:REAL;

BEGIN
RMAXINT:=MAXINT;
WRITELN(' VECTOR INPUT FOR ',PROMPT);
WRITELN('  INPUT THE ',K:2,' DIMENSIONAL DATA ITEM VECTOR.');
WRITELN('  USE A VALUE <0 TO ABORT INPUT SEQUENCE.');
WRITELN('     ALL INPUTS SHOULD BE');
WRITELN('         0<= X(I) <1');
INDEX:=1;
WHILE (INDEX<=K) DO
     BEGIN                (* ONE FOR EACH DIMENSION. *)
     WRITE('    X(',INDEX:2,') = ');
     READLN(DATAVALUE);
     IF (DATAVALUE>= 0.0) THEN
          BEGIN           (* NO ABORT. *)
          IF (DATAVALUE>= 1.0) THEN
               IDVAL:=MAXINT        (* LARGEST INTEGER. *)
          ELSE
               IDVAL:=TRUNC((RMAXINT+1.0)*DATAVALUE);
                          (* NOW IN [ 0 , MAXINT) RANGE. *)
          DATVEC(. INDEX .):=IDVAL;
          INDEX:=INDEX+1;
          END
     ELSE
          BEGIN           (* ABORT INPUT SEQUENCE. *)
          INDEX:=INDEX+10; (* ESCAPE LOOP. *)
          DATVEC(. 1 .):= -1; (* INDICATE BAD VALUE. *)
          END;
     END;
IF (VECFLG) THEN OUTVEC(DATVEC);
END;   (* OF INPDIV *)


BEGIN          (* ADAMTEST *)
(*$NULLBODY*)
END.
```

```
PROGRAM ADAMTEST;
     (* THIS PROGRAM TESTS THE ADAM PROCEDURES *)

CONST
     MAXNODES = 50;     (* SMALL BUFFER FOR NOW. *)
     MAXDIM = 10;       (* MAX NUMBER OF DIMENSIONS. *)
     MAXBIT = 16;       (* INTEGER LENGTH IN BITS. *)
          (* MAXBIT SHOULD ALWAYS BE BIGGER THAN MAXDIM. *)
     MAXDI = 100;        (* MAX DATA ITEMS IN A RETRIEVAL. *)

TYPE NODECELL = RECORD
          SPPTR,CDPTR:INTEGER;
          DATA:ARRAY (. 1..2 .) OF INTEGER;
          END;
     NODEARRAY = ARRAY (. 0..MAXNODES .) OF NODECELL;
     MAPDIRECT = RECORD
          NUMDIM,NUMREG,NUMNODES,NUMLEV:INTEGER;
          END;
     ADAMMAP = RECORD
          DIRECT:MAPDIRECT;
          NODE:NODEARRAY;
          END;
     PFILE = TEXT;
     BITSET = ARRAY (. 1..MAXBIT .) OF BOOLEAN;
     BITARRAY = ARRAY (. 1..MAXBIT .) OF BITSET;
     DIVEC = ARRAY (. 0..MAXBIT .) OF INTEGER;
     PUFLAG = (PACK,UNPACK);
     SCHRES = (NOTDONE,INSERT,MATCH);
     REGDEF = RECORD
          LOWVAL,HIGHVAL:DIVEC;
          END;
     SEQDS = ARRAY (. 0..MAXDI .) OF DIVEC;
     C8ARAY = ARRAY (. 1..8 .) OF CHAR;

VAR  MAP:ADAMMAP;
     MASK:ARRAY (. 1..MAXBIT .) OF INTEGER;
     PRFILE:PFILE;
     PRFLAG:(DISPLAY,PRINTER);
     BITFLG,VECFLG:BOOLEAN;          (* DEBUG FLAGS. *)
```

```
PROCEDURE ADEBUG(VAR MAP:ADAMMAP);
     (* ALLOWS SOME DEBUGGING AIDS TO BE ACTIVATED/DEACTIVATED
     INTERACTIVELY, AND SOME DEBUGGING TOOLS TO BE USED. *)

VAR  CMD:PACKED ARRAY (. 1..4 .) OF CHAR;
```

```
PROCEDURE PRTNODE(VAR OFILE:PFILE;NODE:NODECELL);
     (* PRINTS OUT THE CONTENTS OF A SINGLE NODE. *)
BEGIN
WRITE(OFILE,'  S/P= ',NODE.SPPTR:6);
WRITE(OFILE,'  C/D= ',NODE.CDPTR:6);
WRITE(OFILE,'  DATA=  ',NODE.DATA(. 1 .):4 HEX);
WRITE(OFILE,'  ',NODE.DATA(. 2 .):4 HEX);
WRITELN(OFILE);
END;   (* OF PRTNODE *)
```

```
PROCEDURE OUTDIR(VAR OFILE:PFILE;VAR MAP:ADAMMAP);
     (* OUTPUTS THE MAP DIRECTORY TO OFILE. *)
BEGIN
WITH MAP.DIRECT DO
     BEGIN
     WRITELN(OFILE,' MAP DIRECTORY.');
     WRITELN(OFILE,'  SIZE =',NUMNODES,'  NODES.');
     WRITELN(OFILE,'    ',NUMREG,' REGIONS');
     WRITELN(OFILE,'    ',NUMDIM,' DIMENSIONS');
     WRITELN(OFILE,'    ',NUMLEV,' LEVELS');
     END;
END;   (* OF OUTDIR *)
```

```
PROCEDURE HELP;
     (* PRINTS OUT LEGAL DEBUG COMMANDS. *)
BEGIN
WRITELN('  BITS - - - - SWITCHES BIT OUTPUT FLAG.');
WRITELN('  DIR  - - - - OUTPUT MAP DIRECTORY.');
WRITELN('  DISP - - - - SWITCHES OUTPUT TO DISPLAY.');
WRITELN('  DUMP - - - - PRINTS THE MAP.');
WRITELN('  HELP - - - - DISPLAYS THIS TABLE.');
WRITELN('  LIST - - - - SWITCHES OUTPUT TO PRINTER.');
WRITELN('  PAGE - - - - OUTPUTS A PAGE ON PRINT FILE.');
WRITELN('  STOP - - - - EXIT DEBUG MODE.');
WRITELN('  VECS - - - - SWITCHES VECTOR OUTPUT FLAG.');
END;   (* OF HELP *)
```

```
PROCEDURE DUMPDAT(VAR OFILE:PFILE;VAR MAP:ADAMMAP);
     (* PRINTS OUT THE MAP TO OFILE. *)
VAR  START,STOP,INDEX:INTEGER;

BEGIN
WRITELN('  DUMP NODE CONTENTS.');
WRITE('     INDEX OF STARTING NODE = ');
READLN(START);
WRITE('     INDEX OF LAST NODE = ');
READLN(STOP);
IF (START<0) THEN START:=0;
IF (STOP>MAP.DIRECT.NUMNODES) THEN STOP:=MAP.DIRECT.NUMNODES;
IF (STOP<START) THEN STOP:=START;
WRITELN(OFILE,' DUMP OF ADAM MAP.');
WRITELN(OFILE,'     FROM ',START:5,'     TO ',STOP:5);
INDEX:=START;
WHILE (INDEX<=STOP) DO
     BEGIN            (* ONE NODE AT A TIME. *)
     WRITE(OFILE,'  ',INDEX:5,'    ');
     PRTNODE(OFILE,MAP.NODE(. INDEX .));
     INDEX:=INDEX+1;
     END;
WRITELN(OFILE);
END;    (* OF DUMPDAT *)
```

```
BEGIN    (* DEBUG *)
CMD:='NULL';
WHILE (CMD<>'STOP') DO
     BEGIN
     WRITE('  DEBUG COMMAND = ');
     READLN(CMD);
     IF (CMD='BITS') THEN BITFLG:=NOT BITFLG;
     IF (CMD='DIR ') THEN
          IF (PRFLAG=DISPLAY) THEN OUTDIR(OUTPUT,MAP)
          ELSE OUTDIR(PRFILE,MAP);
     IF (CMD='DISP') THEN PRFLAG:=DISPLAY;
                                   (* DUMP TO SCREEN. *)
     IF (CMD='DUMP') THEN
          IF (PRFLAG=DISPLAY) THEN
               DUMPDAT(OUTPUT,MAP)
          ELSE DUMPDAT(PRFILE,MAP);
     IF (CMD='HELP') THEN HELP;
     IF (CMD='LIST') THEN PRFLAG:=PRINTER;
                                   (* DUMP TO PRINTER. *)
     IF (CMD='PAGE') THEN
          IF (PRFLAG=DISPLAY) THEN PAGE(OUTPUT)
          ELSE PAGE(PRFILE);
     IF (CMD='STOP') THEN WRITELN('  STOPPING');
     IF (CMD='VECS') THEN VECFLG:=NOT VECFLG;
     END;
WRITELN('  END OF DEBUG. ');
END;    (* OF DEBUG *)

BEGIN          (* ADAMTEST *)
(*$NULLBODY*)
END.
```

```
PROGRAM ADAMLIB;
     (* THIS DUMMIED PROGRAM CONTAINS THE ADAM PROCEDURES *)

CONST
     MAXNODES = 50;     (* SMALL BUFFER FOR NOW. *)
     MAXDIM = 10;       (* MAX NUMBER OF DIMENSIONS. *)
     MAXBIT = 16;       (* INTEGER LENGTH IN BITS. *)
          (* MAXBIT SHOULD ALWAYS BE BIGGER THAN MAXDIM. *)
     MAXDI = 100;        (* MAX DATA ITEMS IN A RETRIEVAL. *)

TYPE NODECELL = RECORD
          SPPTR,CDPTR:INTEGER;
          DATA:ARRAY (. 1..2 .) OF INTEGER;
          END;
     NODEARRAY = ARRAY (. 0..MAXNODES .) OF NODECELL;
     MAPDIRECT = RECORD
          NUMDIM,NUMREG,NUMNODES,NUMLEV:INTEGER;
          END;
     ADAMMAP = RECORD
          DIRECT:MAPDIRECT;
          NODE:NODEARRAY;
          END;
     PFILE = TEXT;
     BITSET = ARRAY (. 1..MAXBIT .) OF BOOLEAN;
     BITARRAY = ARRAY (. 1..MAXBIT .) OF BITSET;
     DIVEC = ARRAY (. 0..MAXBIT .) OF INTEGER;
     PUFLAG = (PACK,UNPACK);
     SCHRES = (NOTDONE,INSERT,MATCH);
     REGDEF = RECORD
          LOWVAL,HIGHVAL:DIVEC;
          END;
     SEQDS = ARRAY (. 0..MAXDI .) OF DIVEC;
     C8ARAY = ARRAY (. 1..8 .) OF CHAR;

VAR  MAP:ADAMMAP;
     MASK:ARRAY (. 1..MAXBIT .) OF INTEGER;
     PRFILE:PFILE;
     PRFLAG:(DISPLAY,PRINTER);
     BITFLG,VECFLG:BOOLEAN;          (* DEBUG FLAGS. *)

(*              I/O ROUTINES.        *)

PROCEDURE OUTBITS(OFILE:PFILE;BITS:BITSET;K:INTEGER);
     EXTERNAL;
     (* OUTPUTS K BITS FROM THE BITSET TO OFILE. *)

PROCEDURE OUTVEC(DATVEC:DIVEC);EXTERNAL;
     (* OUTPUTS DATVEC TO PRFILE IN HEX AND INTEGER. *)


(*                FREE NODE MANIPULATION ROUTINES           *)

PROCEDURE GETCELL(VAR NODE:NODEARRAY;VAR NEW:INTEGER);
     EXTERNAL;
     (* RETURNS THE INDEX OF THE NEXT FREE NODE AFTER
     REMOVING IT FROM THE FREE LIST.
               NEW=0 ON RETURN
                    MEANS "OUT OF FREE STORAGE." *)

PROCEDURE NEWBUFF(VAR NODE:NODEARRAY;M:INTEGER); EXTERNAL;
     (* INITIALIZES A DATA BUFFER OF LENGTH M IN THE NODE
     ARRAY INTO A FREE STORAGE LIST.
          THIS SETS UP NODE(0) AS THE HEAD OF THE LIST
     OF FREE NODES.
               NODE(0).SPPTR-----POINTER TO NEXT FREE NODE.
               NODE(0).CDPTR-----POINTER TO LAST FREE NODE.
               NODE(0).DATA(1)---NUMBER OF NODES. *)

PROCEDURE RETCELL(VAR NODE:NODEARRAY;OLD:INTEGER); EXTERNAL;
     (* RETURNS AN OLD NODE TO THE FREE STORAGE LIST
     AFTER CHECKING FOR DETECTABLE ERRORS. *)

(*             BIT MANIPULATION UTILITIES.             *)


PROCEDURE BITPU(VAR BITS:BITSET;VAR INTWD:INTEGER;DIR:PUFLAG);
     EXTERNAL;
     (* KEYS ON DIR (PACK OR UNPACK) TO
               PACK BITS INTO INTWD, OR
               UNPACK INTWD INTO BITS.  *)

PROCEDURE CRTLSK(VAR LSKVEC:DIVEC;DATVEC:DIVEC;K:INTEGER);
     EXTERNAL;
     (* CREATE THE LEVEL SEARCH KEYS FROM THE DATA ITEM VECTOR,
     DATVEC.  THE ITH LEVEL SEARCH KEY CONSISTS OF THE ITH MSB
     OF EACH OF THE ELEMENTS OF DATVEC.  THERE ARE K ELEMENTS
     IN DATVEC, AND MAXBIT ELEMENTS IN LSKVEC. *)


(*             MAP MANIPULATION UTILITIES.             *)

PROCEDURE NODEINS(VAR MAP:ADAMMAP;NEWONE,PARENT,LASTONE,
     NEXTONE:INTEGER); EXTERNAL;
     (* INSERTS THE NODE AT NEWONE BETWEEN LASTONE AND NEXTONE,
     COMPENSATING IF EITHER IS A PARENT. *)

PROCEDURE MAPSRCH(VAR LSEARCH:SCHRES;VAR PARENT,LASTONE,
     NEXTONE,LEVEL:INTEGER;VAR MAP:ADAMMAP;
     VAR LSK:DIVEC); EXTERNAL;
     (* SEARCHES THE MAP IN NODEARRAY FOR THE LEVEL SEARCH
     KEYS IN LSK.  LSEARCH RETURNS INSERT OR MATCH, AND THE
     POSITION IS RETURNED IN LEVEL, PARENT, LASTONE, AND
     NEXTONE. *)
```

```
PROCEDURE AADD(VAR MAP:ADAMMAP;DATVEC:DIVEC);
     (* ADDS DATA POINTS TO THE MAP.  USES INTERACTIVE DATA ENTRY. *)
VAR  LSKVEC:DIVEC;
     NEWONE,PARENT,LASTONE,NEXTONE,BOTTOM,LEVEL:INTEGER;
                                    (* NODE PTRS. *)
     LSEARCH:SCHRES;
```

```
PROCEDURE BUILDB(VAR MAP:ADAMMAP;VAR LEVEL,NEWONE,BOTTOM:
     INTEGER;VAR LSKVEC:DIVEC);
     (* BUILDS A BRANCH IN MAP, STARTING AT LEVEL, USING
     KEYS FROM LSKVEC, AND EXTENDING TO THE "DIRECT.NUMLEV"
     LEVEL. *)
VAR  L,INDEX,LASTNODE,NEWNODE:INTEGER;

BEGIN
WITH MAP DO
     BEGIN
     L:=DIRECT.NUMLEV;    (* MAX DEPTH OF THIS MAP. *)
     INDEX:=LEVEL;
     GETCELL(NODE,NEWNODE);
     NEWONE:=NEWNODE;
     IF (NEWONE>0) THEN
          NODE(.NEWONE.).DATA(. 1 .):=LSKVEC(.INDEX.);
     WHILE(INDEX<L) AND (NEWONE>0) DO
          BEGIN                    (* ADD NODE LEVELS. *)
          LASTNODE:=NEWNODE;
          INDEX:=INDEX+1;
          GETCELL(NODE,NEWNODE);
          IF (NEWNODE>0) THEN
               BEGIN               (* VALID CELL. *)
               NODEINS(MAP,NEWNODE,LASTNODE,LASTNODE,0);
                                   (* INSERTS BELOW IT. *)
               NODE(.NEWNODE.).DATA(. 1 .):=LSKVEC(.INDEX.);
                                   (* STORE KEY. *)
               END
          ELSE
               BEGIN               (* NOT ENOUGH CELLS. *)
               LASTNODE:=NEWONE;
               WHILE (LASTNODE>0) DO
                    BEGIN          (* RETURN THE NODES. *)
                    NEWNODE:=NODE(.LASTNODE.).CDPTR;
                    RETCELL(NODE,LASTNODE);
                    LASTNODE:=NEWNODE;
                    END;
               NEWONE:=NEWNODE;         (* NULL POINTER. *)
               END;
          END;
     END;
BOTTOM:=NEWNODE;
END;   (* OF BUILDB *)
```

```
BEGIN  (* AADD *)
IF (DATVEC(. 1 .) >= 0) THEN
     BEGIN                (* DATA VALID. *)
     CRTLSK(LSKVEC,DATVEC,MAP.DIRECT.NUMDIM);
                          (* CREATE LEVEL SEARCH KEYS. *)
     IF (VECFLG) THEN OUTVEC(LSKVEC);
     MAPSRCH(LSEARCH,PARENT,LASTONE,NEXTONE,LEVEL,
          MAP,LSKVEC);
                          (* SEARCH THE MAP FOR INSERT POINT OR
                          MATCH. *)
     IF (LSEARCH = INSERT) THEN
          BEGIN           (* BUILD A BRANCH TO INSERT. *)
          BUILDB(MAP,LEVEL,NEWONE,BOTTOM,LSKVEC);
          IF (BOTTOM>0) THEN
               BEGIN      (* ONLY VALID IF BRANCH IS GOOD. *)
               NODEINS(MAP,NEWONE,PARENT,LASTONE,NEXTONE);
                                    (* AND INSERT IT. *)
               MAP.NODE(.BOTTOM.).CDPTR:=-DATVEC(.0.);
                                    (* THE DATA POINT VALUE. *)
               END;
          END
     ELSE
          IF(LSEARCH = MATCH) THEN
               WRITELN(' *** DUPLICATE DATA POINT. ***');
     END;
END;   (* OF AADD *)
```

```
PROCEDURE ACREATE(VAR MAP:ADAMMAP;M,K,L:INTEGER);
     (*CREATES AN EMPTY ADAM MAP IN THE M-NODE MAP BUFFER AREA,
     SETTING UP THE DIRECTORY AND FREE STORAGE LIST. *)
VAR  ROOT:INTEGER;

BEGIN
WITH MAP DO
     BEGIN                (* INITIALIZE THE MAP. *)
     IF (M<1) THEN NODE(. 0 .).SPPTR:=0
     ELSE
          BEGIN
          NEWBUFF(NODE,M); (* INITIALIZE THE FREE NODE LIST.*)
          DIRECT.NUMDIM:=K; (* NUMBER OF DIMENSIONS. *)
          DIRECT.NUMNODES:=M; (* MAX NUMBER OF NODES. *)
          DIRECT.NUMREG:=0;   (* NO REGIONS INITIALLY. *)
          DIRECT.NUMLEV:=L;   (* MAX NUMBER OF LEVELS. *)
          GETCELL(NODE,ROOT); (* GET ROOT NODE. *)
          WITH NODE(.ROOT.) DO
               BEGIN
               SPPTR:=-1;   (* POINTS TO SELF AS PARENT. *)
               CDPTR:=0;    (* NO CHILDREN YET. *)
               DATA(. 1 .):=0; (* KEY:=0 *)
               DATA(. 2 .):=0; (* NO REGIONS. *)
               END;
          END;
     END;
END;   (* OF CREATE *)

BEGIN            (* ADAMTEST *)
(*$NULLBODY*)
END.
```

```
PROGRAM ADAMLIB2;
     (* THIS DUMMIED PROGRAM CONTAINS THE ADAM PROCEDURES
     RETRIEVE, FIND, AND DELETE. *)

CONST
     MAXNODES = 50;      (* SMALL BUFFER FOR NOW. *)
     MAXDIM = 10;        (* MAX NUMBER OF DIMENSIONS. *)
     MAXBIT = 16;        (* INTEGER LENGTH IN BITS. *)
          (* MAXBIT SHOULD ALWAYS BE BIGGER THAN MAXDIM. *)
     MAXDI = 100;         (* MAX DATA ITEMS IN A RETRIEVAL. *)
     MAXREG = 8;           (* MAX NUMBER OF FLAGGED REGIONS. *)

TYPE NODECELL = RECORD
          SPPTR,CDPTR:INTEGER;
          DATA:ARRAY (. 1..2 .) OF INTEGER;
          END;
     NODEARRAY = ARRAY (. 0..MAXNODES .) OF NODECELL;
     MAPDIRECT = RECORD
          NUMDIM,NUMREG,NUMNODES,NUMLEV:INTEGER;
          END;
     ADAMMAP = RECORD
          DIRECT:MAPDIRECT;
          NODE:NODEARRAY;
          END;
     POSITION = RECORD
          PARENT,CURRENT,LASTONE,NEXTONE,REGION:INTEGER;
          END;
     MAPPOS = ARRAY (. 1..MAXBIT .) OF POSITION;
     PFILE = TEXT;
     BITSET = ARRAY (. 1..MAXBIT .) OF BOOLEAN;
     BITARRAY = ARRAY (. 1..MAXBIT .) OF BITSET;
     DIVEC = ARRAY (. 0..MAXBIT .) OF INTEGER;
     PUFLAG = (PACK,UNPACK);
     SCHRES = (NOTDONE,INSERT,MATCH);
     REGCOND = (INSIDE,OVERLAP,OUTSIDE,UNKNOWN);
     MAPSTAT = (OKAY,ENDLEV,TERM,EMPTY,ATTOP,ATBOT);
     MAPOPER = (UP,DOWN,ACROSS,STOP,COMPARE);
     TRAVRES = (TOP,LEAF,NEWNODE,OLDNODE,NEXT);
     REGDEF = RECORD
          LOWVAL,HIGHVAL:DIVEC;
          END;
     SEQDS = ARRAY (. 0..MAXDI .) OF DIVEC;
     C8ARAY = ARRAY (. 1..8 .) OF CHAR;

VAR  MAP:ADAMMAP;
     MASK:ARRAY (. 1..MAXBIT .) OF INTEGER;
     PRFILE:PFILE;
     PRFLAG:(DISPLAY,PRINTER);
     BITFLG,VECFLG:BOOLEAN;          (* DEBUG FLAGS. *)

(*              I/O ROUTINES.          *)

PROCEDURE OUTBITS(OFILE:PFILE;BITS:BITSET;K:INTEGER);
     EXTERNAL;
     (* OUTPUTS K BITS FROM THE BITSET TO OFILE. *)

(*                FREE NODE MANIPULATION ROUTINES         *)

PROCEDURE RETCELL(VAR NODE:NODEARRAY;OLD:INTEGER); EXTERNAL;
     (* RETURNS AN OLD NODE TO THE FREE STORAGE LIST
     AFTER CHECKING FOR DETECTABLE ERRORS. *)

(*             BIT MANIPULATION UTILITIES.            *)

PROCEDURE BITPU(VAR BITS:BITSET;VAR INTWD:INTEGER;DIR:PUFLAG);
     EXTERNAL;
     (* KEYS ON DIR (PACK OR UNPACK) TO
               PACK BITS INTO INTWD, OR
               UNPACK INTWD INTO BITS.  *)

(*             MAP COMPARISON ROUTINES.          *)

PROCEDURE ARSELECT(REGFLD,REGIND:INTEGER;VAR COND:REGCOND);
     EXTERNAL;
     (* EXTRACTS THE CONDITION OF THE REGION FLAGS FOR THE
     REGIND REGION, USING THE REGION FLAGS IN REGFLD.  RETURNS
     THE CONITION OF THE FLAGS IN COND.
     THE REGION FLAGS ARE TWO BITS, AND ARE INTERPRETED AS
     FOLLOWS:
               00---UNKNOWN
               01---OUTSIDE
               10---INSIDE
               11---OVERLAP
     IF THE REGION INDEX IS ILLEGAL THEN THE RETURN IS ALWAYS
     "INSIDE".  *)

(*             MAP MANIPULATION UTILITIES.            *)

PROCEDURE AMAPTRAV(VAR MAP:ADAMMAP;VAR POS:MAPPOS;
     VAR LEVEL:INTEGER;OPER:MAPOPER;VAR RESULT:TRAVRES;
     VAR PRFILE:PFILE);
     EXTERNAL;
     (* TRAVERSES THE ADAM MAP ONE NODE AT A TIME IN
     THE DIRECTIONS INDICATED BY OPER.  KEEPS TRACK OF THE
     POSITION IN POS AND LEVEL, AND RETURNS THE RESULTING NODE
     TYPE AS FOLLOWS:
               TOP --- DONE, ERROR OR NOT
               LEAF--- REACHED A LEAF
               NEWNODE--- NEW NODE- OLD LEVEL, OR
                          NEW NODE- NEW LEVEL
               OLDNODE--- OLD NODE- OLD LEVEL  *)
```

```
PROCEDURE ADELETE(VAR MAP:ADAMMAP;INDEX:INTEGER);
     (* DELETES THE DATA POINTS IN THE FIND REGION INDICATED BY
     REGIND IN THE MAP. *)
TYPE REMRES = (LAST,NOTLAST,YES,NO);
VAR  RESULT:TRAVRES;
     OPER:MAPOPER;
     REMFLG:REMRES;
     COND:REGCOND;
     NEWREG,POINT,LEVEL:INTEGER;
     POS:MAPPOS;
```

```
PROCEDURE REMNODE(VAR MAP:ADAMMAP;VAR POSIT:POSITION;
     VAR REMFLG:REMRES);
     (* REMOVES A SINGLE NODE FROM THE MAP. ITS LOCATION IS IN
     POSIT, WHICH IS UPDATED TO THE NEXT CELL.  THE REMOVED
     NODE IS RETURNED TO FREE STORAGE. *)
BEGIN
WITH POSIT DO
     BEGIN
     POINT:=CURRENT;
     IF (PARENT<>LASTONE) THEN
          MAP.NODE(.LASTONE.).SPPTR:=NEXTONE
     ELSE IF(NEXTONE<0) THEN MAP.NODE(.PARENT.).CDPTR:=0
          ELSE MAP.NODE(.PARENT.).CDPTR:=NEXTONE;
     IF (NEXTONE<0) THEN REMFLG:=LAST
     ELSE BEGIN
          REMFLG:=NOTLAST;
          CURRENT:=NEXTONE;
          NEXTONE:=MAP.NODE(.CURRENT.).SPPTR;
          END;
     END;
RETCELL(MAP.NODE,POINT);
END;   (* OF REMNODE *)
```

```
BEGIN   (* ADELETE *)
LEVEL:=1;
WITH POS(.LEVEL.) DO
     BEGIN
     CURRENT:=1;
     PARENT:=1;
     NEXTONE:=MAP.NODE(.CURRENT.).SPPTR;
     LASTONE:=1;
     REGION:=INDEX;
     NEWREG:=REGION;
     END;
RESULT:=NEWNODE;
WHILE (RESULT<>TOP) DO
     BEGIN
     WITH POS(.LEVEL.) DO
          BEGIN
          IF (OPER=DOWN) AND (RESULT<>NEXT) THEN
               REGION:=NEWREG;
          CASE RESULT OF
               NEWNODE:BEGIN
                    RESULT:=NEXT;
                    ARSELECT(MAP.NODE(.CURRENT.).DATA(.2.),
                         REGION,COND);
                    IF (COND=OUTSIDE) THEN OPER:=ACROSS
                    ELSE BEGIN      (* TRAVERSE THE SUBTREE. *)
                         IF (COND=INSIDE) THEN NEWREG:=0
                         ELSE NEWREG:=REGION;
                         OPER:=DOWN;
                         END;
                    END;
               NEXT:                (* PERFORM NEXT MOVE. *)
                    AMAPTRAV(MAP,POS,LEVEL,OPER,RESULT,PRFILE);
               OLDNODE:             (* MOVING UP, REVISITED. *)
                    IF (MAP.NODE(.CURRENT.).CDPTR=0) THEN
                         REMFLG:=YES  (* UNUSED, REMOVE IT. *)
                    ELSE BEGIN      (* USED, MOVE ON. *)
                         RESULT:=NEXT;
                         OPER:=ACROSS;
                         END;
               LEAF:REMFLG:=YES;  (* LEAF INSIDE, REMOVE IT. *)
               END;
          IF (REMFLG=YES) THEN
               BEGIN      (* REMOVE THE CURRENT NODE. *)
               REMNODE(MAP,POS(.LEVEL.),REMFLG);
               IF (REMFLG=LAST) THEN
                    BEGIN         (* NO MORE AT LOWER LEVEL. *)
                    LEVEL:=LEVEL-1;
                    RESULT:=OLDNODE;
                    NEWREG:=POS(.LEVEL.).REGION;
                    END
               ELSE RESULT:=NEWNODE;
               REMFLG:=NO;
               END;
          END;
          END;
      END;   (* OF ADELETE *)
```

```
PROCEDURE AFIND(VAR MAP:ADAMMAP;INDEX:INTEGER;
     VAR RD:REGDEF);
     (* FINDS A REGION IN THE ADAM MAP AND FLAGS THE SUBTREES
     AS INSIDE, OUTSIDE, OR OVERLAPPING THE REGION.*)

VAR  TRACE:BITARRAY;      (* COLLECTS THE TRACE BIT PATTERNS. *)
     I,K,LEVEL,CZERO,VALUE:INTEGER;
     POS:MAPPOS;
     RESULT:TRAVRES;
     OPER:MAPOPER;
     COND:REGCOND;
     LFLAG:BOOLEAN;
```

```
PROCEDURE AREGCOMP(KEYFLD,K,LEVEL:INTEGER;
     VAR TRACE:BITARRAY;VAR RD:REGDEF;VAR COND:REGCOND);
     (* COMPARES A RECTANGULAR SEARCH REGION AND A NODE
     REGION TO DETERMINE THEIR INTERSECTION CONDITION.
     THE RESULTS ARE AS FOLLOWS:
                 OUTSIDE  - - -NODE REGION IS OUTSIDE SEARCH
                 INSIDE   - - -NODE REGION IS ENTIRELY INSIDE
                 OVERLAP  - - -NODE AND SEARCH REGIONS OVERLAP
                                 BUT THE NODE IS NOT ENTIRELY
                                 INSIDE THE SEARCH
                 UNKNOWN  - - -NO COMPARISON HAS BEEN DONE. *)
VAR I,SIZE,NRLOW,NRHIGH,RLOW,RHIGH,CVALUE:INTEGER;
     KEY:BITSET;

BEGIN
IF (LEVEL>1) THEN SIZE:=MASK(.MAXBIT-LEVEL+1.)
ELSE SIZE:=0;
IF (VECFLG) THEN
     BEGIN
     WRITELN(PRFILE);
     WRITELN(PRFILE,'  REGION LIMITS FROM FIND REGION',
              ' COMPARE.');
     END;
BITPU(KEY,KEYFLD,UNPACK);
COND:=INSIDE;
I:=1;
WHILE (I<=K) DO
     BEGIN  (* CHECK EACH DIMENSION. *)
     TRACE(.I,LEVEL.):=KEY(.MAXBIT-I+1.);
     BITPU(TRACE(.I.),CVALUE,PACK);
     IF (LEVEL<=1) THEN
          BEGIN
          NRLOW:=0;
          NRHIGH:=MAXINT;
          END
     ELSE
          BEGIN              (* SET UPPER AND LOWER INTERVAL
                                 LIMITS FOR COMPARISON.*)
          NRLOW:=CVALUE;
          NRHIGH:=(CVALUE-1)+SIZE;
          END;
     RLOW:=RD.LOWVAL(.I.);
     RHIGH:=RD.HIGHVAL(.I.);
     IF (VECFLG) THEN WRITELN(PRFILE,NRLOW,NRHIGH,RLOW,RHIGH);
     IF (NRHIGH<RLOW) OR (RHIGH<NRLOW) THEN
          COND:=OUTSIDE           (* MUTUALLY EXCLUSIVE. *)
     ELSE IF (NRLOW<RLOW) OR (RHIGH<NRHIGH) THEN
          COND:=OVERLAP;
     IF (COND=OUTSIDE) THEN I:=K+1  (* ESCAPE. *)
     ELSE I:=I+1;         (* NEXT DIMENSION. *)
     END;
IF (VECFLG) THEN WRITELN(PRFILE);
END;   (* OF AREGCOMP *)
```

```
PROCEDURE ARSET(VAR REGFLD:INTEGER;REGIND:INTEGER;
     COND:REGCOND);
     (* SETS THE BITS FOR THE REGION INDICATION IN THE
     PROPER LOCATION FOR THE REGION INDEX IN THE REGION
     FIELD.  THE REGION FLAGS ARE AS FOLLOWS:
               00---UNKNOWN
               01---OUTSIDE
               10---INSIDE
               11---OVERLAP.
     PERFORMS UNION WITH PREVIOUSLY SELECTED PORTIONS OF THE
     REGION.       *)
VAR  REGION:BITSET;
     BITLOC:INTEGER;
BEGIN
IF (REGIND>=1) AND (REGIND<=MAXREG) THEN
     BEGIN
     BITLOC:=(REGIND-1)*2+1;
     BITPU(REGION,REGFLD,UNPACK);
     REGION(.BITLOC.):= (COND=INSIDE) OR (COND=OVERLAP) OR
          REGION(.BITLOC.);
     REGION(.BITLOC+1.):=(COND=OUTSIDE) OR (COND=OVERLAP);
     IF (BITFLG) THEN
          BEGIN
          WRITELN(PRFILE,' NEW REGION FLAG FIELD.');
          OUTBITS(PRFILE,REGION,MAXBIT);
          END;
     BITPU(REGION,REGFLD,PACK);
     END;
END;   (* OF ARSET *)
```

```
BEGIN   (* FIND *)
CZERO:=0;
K:=MAP.DIRECT.NUMDIM;
I:=1;
WHILE (I<=MAXBIT) DO
     BEGIN              (* INIT TRACE TO 0. *)
     BITPU(TRACE(.I.),CZERO,UNPACK);
     I:=I+1;
     END;
LEVEL:=1;
WITH POS(.LEVEL.) DO
     BEGIN              (* SET POSITION OF ROOT. *)
     CURRENT:=1;
     PARENT:=1;
     NEXTONE:=MAP.NODE(.CURRENT.).SPPTR;
     LASTONE:=1;
     END;
LFLAG:=FALSE;
RESULT:=NEWNODE;
WHILE (RESULT<>TOP) DO
     BEGIN         (* TRAVERSE THE TREE AS NEEDED. *)
     WITH POS(.LEVEL.) DO
          BEGIN    (* BASE POSITION ON THE CURRENT LEVEL
                   FOR EACH PASS IN THIS LOOP. *)
          CASE RESULT OF
               NEWNODE:BEGIN  (* COMPARE CURRENT NODE. *)
                    AREGCOMP(MAP.NODE(.CURRENT.).DATA(.1.),
                         K,LEVEL,TRACE,RD,COND);
                    ARSET(MAP.NODE(.CURRENT.).DATA(.2.),
                         INDEX,COND);
                    IF LFLAG THEN OPER:=ACROSS
                    ELSE IF (COND=OVERLAP) THEN OPER:=DOWN
                    ELSE OPER:=ACROSS;
                    RESULT:=NEXT;
                    END;
               NEXT:BEGIN  (* MOVE TO NEXT NODE. *)
                    LFLAG:=FALSE;
                    AMAPTRAV(MAP,POS,LEVEL,OPER,RESULT,PRFILE);
                    IF (RESULT=OLDNODE) THEN
                         BEGIN    (* RETURNED UP TO NODE
                                  ALREADY VISITED. *)
                         I:=1;
                         WHILE (I<=K) DO
                              BEGIN    (* CLEAR OUT LAST
                                       LEVEL OF TRACE. *)
                              TRACE(.I,LEVEL+1.):=FALSE;
                              I:=I+1;
                              END;
                         OPER:=ACROSS;
                         RESULT:=NEXT;
                         END;
                    END;
               LEAF:BEGIN
                    IF (BITFLG) THEN
```

```
                         BEGIN
                         WRITELN(PRFILE,' TRACE VALUES');
                         I:=1;
                         WHILE (I<=K) DO
                              BEGIN
                              BITPU(TRACE(.I.),VALUE,
                                   PACK);
                              WRITE(PRFILE,
                                   (VALUE/(MAXINT+1.0)):7:5,
                                   '  ');
                              I:=I+1;
                              END;
                         WRITELN(PRFILE);
                         END;
                    LFLAG:=TRUE;
                    RESULT:=NEWNODE;
                    END;
               END;
          END;
     END;
END;   (* OF AFIND *)
```

```
PROCEDURE ARETRIEVE(VAR MAP:ADAMMAP;INDEX:INTEGER;
     VAR SDS:SEQDS);
     (* RETRIEVES THE REGION DEFINED BY INDEX INTO THE
     SEQUENTIAL DATA SET SDS. *)
VAR  RESULT:TRAVRES;
     POS:MAPPOS;
     KEY:BITSET;
     COUNT,I,K,LASTREG,LEVEL,NEWREG:INTEGER;
     TRACE:BITARRAY;
     OPER:MAPOPER;
     COND:REGCOND;

BEGIN
LEVEL:=1;
K:=MAP.DIRECT.NUMDIM;
COUNT:=0;
WITH POS(.LEVEL.) DO
     BEGIN
     CURRENT:=1;
     PARENT:=1;
     NEXTONE:=MAP.NODE(.CURRENT.).SPPTR;
     LASTONE:=1;
     REGION:=INDEX;
     NEWREG:=REGION;
     END;
OPER:=DOWN;
RESULT:=NEWNODE;
WHILE (RESULT<>TOP) DO
     BEGIN               (* TRAVERSE ENTIRE TREE. *)
     WITH POS(.LEVEL.) DO
          BEGIN
          IF (OPER=DOWN) AND (RESULT<>NEXT) THEN
               REGION:=NEWREG;
          CASE RESULT OF
               NEWNODE: BEGIN      (* COMPARE CURRENT NODE. *)
                    RESULT:=NEXT;
                    ARSELECT(MAP.NODE(.CURRENT.).DATA(.2.),
                         REGION,COND);
                    IF (COND=INSIDE) OR
                         (COND=OVERLAP) THEN
                         BEGIN     (* TRAVERSE THE SUBTREE. *)
                         BITPU(KEY,
                              MAP.NODE(.CURRENT.).DATA(.1.),
                              UNPACK);
                         I:=1;
                         WHILE (I<=K) DO
                              BEGIN          (* SAVE THE TRACE
                                             BITS. *)
                              TRACE(.I,LEVEL.):=
                                   KEY(.MAXBIT-I+1.);
                              I:=I+1;
                              END;
                         IF (COND=INSIDE) THEN NEWREG:=0
                         ELSE NEWREG:=REGION;
```

```
                                OPER:=DOWN;
                                END
                            ELSE OPER:=ACROSS; (* IGNORE SUBTREE. *)
                            END;
                    NEXT:BEGIN      (* PERFORM NEXT MOVE. *)
                            AMAPTRAV(MAP,POS,LEVEL,OPER,RESULT,PRFILE);
                            IF (RESULT=OLDNODE) THEN
                                BEGIN  (* CAME UP TO NODE ALREADY
                                          VISITED, MOVE ON. *)
                                RESULT:=NEXT;
                                OPER:=ACROSS;
                                END;
                            END;
                    LEAF:BEGIN      (* SAVE TRACES FOR DATA
                                       ITEM. *)
                            IF (COUNT<MAXDI) THEN COUNT:=COUNT+1;
                            I:=1;
                            WHILE (I<=K) DO
                                BEGIN     (* GET THE TRACE. *)
                                BITPU(TRACE(.I.),SDS(.COUNT,I.),
                                    PACK);
                                I:=I+1;
                                END;
                            SDS(.COUNT,K+1.):=
                                -MAP.NODE(.CURRENT.).CDPTR;
                            RESULT:=NEXT;
                            OPER:=ACROSS;
                            END;
                    END;
                END;
            END;
        SDS(.0,0.):=COUNT;
        END;       (* OF RETRIEVE *)

        BEGIN           (* ADAMTEST *)
        (*$NULLBODY*)
        END.
```

```
PROGRAM ADAMLIB3;
     (* THIS DUMMIED PROGRAM CONTAINS SOME OF THE ADAM
     SUPPORT PROCEDURES *)

CONST
     MAXNODES = 50;     (* SMALL BUFFER FOR NOW. *)
     MAXDIM = 10;       (* MAX NUMBER OF DIMENSIONS. *)
     MAXBIT = 16;       (* INTEGER LENGTH IN BITS. *)
          (* MAXBIT SHOULD ALWAYS BE BIGGER THAN MAXDIM. *)
     MAXDI = 100;        (* MAX DATA ITEMS IN A RETRIEVAL. *)
     MAXREG = 8;          (* MAX NUMBER OF FLAGGED REGIONS. *)

TYPE NODECELL = RECORD
          SPPTR,CDPTR:INTEGER;
          DATA:ARRAY (. 1..2 .) OF INTEGER;
          END;
     NODEARRAY = ARRAY (. 0..MAXNODES .) OF NODECELL;
     MAPDIRECT = RECORD
          NUMDIM,NUMREG,NUMNODES,NUMLEV:INTEGER;
          END;
     ADAMMAP = RECORD
          DIRECT:MAPDIRECT;
          NODE:NODEARRAY;
          END;
     POSITION = RECORD
          PARENT,CURRENT,LASTONE,NEXTONE,REGION:INTEGER;
          END;
     MAPPOS = ARRAY (. 1..MAXBIT .) OF POSITION;
     PFILE = TEXT;
     BITSET = ARRAY (. 1..MAXBIT .) OF BOOLEAN;
     BITARRAY = ARRAY (. 1..MAXBIT .) OF BITSET;
     DIVEC = ARRAY (. 0..MAXBIT .) OF INTEGER;
     PUFLAG = (PACK,UNPACK);
     SCHRES = (NOTDONE,INSERT,MATCH);
     REGCOND = (INSIDE,OVERLAP,OUTSIDE,UNKNOWN);
     MAPSTAT = (OKAY,ENDLEV,TERM,EMPTY,ATTOP,ATBOT);
     MAPOPER = (UP,DOWN,ACROSS,STOP,COMPARE);
     TRAVRES = (TOP,LEAF,NEWNODE,OLDNODE,NEXT);
     REGDEF = RECORD
          LOWVAL,HIGHVAL:DIVEC;
          END;
     SEQDS = ARRAY (. 0..MAXDI .) OF DIVEC;
     C8ARAY = ARRAY (. 1..8 .) OF CHAR;

(*             I/O ROUTINES.        *)

PROCEDURE OUTBITS(OFILE:PFILE;BITS:BITSET;K:INTEGER);
     EXTERNAL;
     (* OUTPUTS K BITS FROM THE BITSET TO OFILE. *)

PROCEDURE OUTVEC(DATVEC:DIVEC);EXTERNAL;
     (* OUTPUTS DATVEC TO PRFILE IN HEX AND INTEGER. *)
(*                  FREE NODE MANIPULATION ROUTINES           *)

PROCEDURE RETCELL(VAR NODE:NODEARRAY;OLD:INTEGER); EXTERNAL;
     (* RETURNS AN OLD NODE TO THE FREE STORAGE LIST
     AFTER CHECKING FOR DETECTABLE ERRORS. *)

(*               BIT MANIPULATION UTILITIES.             *)


PROCEDURE BITPU(VAR BITS:BITSET;VAR INTWD:INTEGER;DIR:PUFLAG);
     EXTERNAL;
     (* KEYS ON DIR (PACK OR UNPACK) TO
               PACK BITS INTO INTWD, OR
               UNPACK INTWD INTO BITS.  *)

PROCEDURE CRTLSK(VAR LSKVEC:DIVEC;DATVEC:DIVEC;K:INTEGER);
     EXTERNAL;
     (* CREATE THE LEVEL SEARCH KEYS FROM THE DATA ITEM VECTOR,
     DATVEC.  THE ITH LEVEL SEARCH KEY CONSISTS OF THE ITH MSB
     OF EACH OF THE ELEMENTS OF DATVEC.  THERE ARE K ELEMENTS
     IN DATVEC, AND MAXBIT ELEMENTS IN LSKVEC. *)
```

```
PROCEDURE ARSELECT(REGFLD,REGIND:INTEGER;VAR COND:REGCOND);
     (* EXTRACTS THE CONDITION OF THE REGION FLAGS FOR THE
     REGIND REGION, USING THE REGION FLAGS IN REGFLD.  RETURNS
     THE CONITION OF THE FLAGS IN COND.
     THE REGION FLAGS ARE TWO BITS, AND ARE INTERPRETED AS
     FOLLOWS:
               00---UNKNOWN
               01---OUTSIDE
               10---INSIDE
               11---OVERLAP
     IF THE REGION INDEX IS ILLEGAL THEN THE RETURN IS ALWAYS
     "INSIDE".  *)
VAR  REGION:BITSET;
     BITLOC:INTEGER;
BEGIN
IF (REGIND<1) OR (REGIND>MAXREG) THEN COND:=INSIDE
ELSE BEGIN
     BITPU(REGION,REGFLD,UNPACK);   (* GET THE BITS. *)
     BITLOC:=(REGIND-1)*2+1;     (* FIND THE FIELD LOCATION. *)
     IF (REGION(.BITLOC.) AND REGION(.BITLOC+1.)) THEN
          COND:=OVERLAP
     ELSE IF REGION(.BITLOC.) THEN COND:=INSIDE
     ELSE IF REGION(.BITLOC+1.) THEN COND:=OUTSIDE
     ELSE COND:=UNKNOWN;
     END;
END;   (* OF ARSELECT *)
```

```
PROCEDURE AMOVE(VAR MAP:ADAMMAP;VAR POS:MAPPOS;
     VAR LEVEL:INTEGER;DIR:MAPOPER;VAR STATUS:MAPSTAT);
     (* MOVES OVER ONE NODE IN THE DIRECTION INDICATED
     BY DIR.  UPDATES THE POSITION STACK, AND RETURNS
     THE STATUS VALUE ACCORDING TO THE NODE TYPE REACHED
     AS FOLLOWS:
               ATTOP -----TOP OF MAP STRUCTURE REACHED
               ATBOT -----BOTTOM REACHED, BUT NO TERMINAL
               EMPTY -----NULL POINTER TO LOWER LEVEL
               ENDLEV ----END OF LEVEL REACHED
               TERM ------BOTTOM REACHED, TERMINAL NODE
               OKAY ------ANY OTHER CONDITION.  *)
VAR  NEWONE:INTEGER;
BEGIN
WITH POS(.LEVEL.) DO
     BEGIN
     CASE DIR OF
          UP:BEGIN
               IF (LEVEL<=1) THEN STATUS :=ATTOP
               ELSE IF (NEXTONE=-CURRENT) THEN STATUS:=ATTOP
               ELSE BEGIN
                    LEVEL:=LEVEL-1;
                    STATUS:=OKAY;
                    END;
               END;
          DOWN:BEGIN
               IF (MAP.NODE(.CURRENT.).CDPTR<0) THEN
                    STATUS:=TERM
               ELSE IF (MAP.NODE(.CURRENT.).CDPTR=0) THEN
                    STATUS:=EMPTY
               ELSE IF (LEVEL>=MAP.DIRECT.NUMLEV) THEN
                    STATUS:=ATBOT
               ELSE BEGIN
                    LEVEL:=LEVEL+1;
                    POS(.LEVEL.).PARENT:=CURRENT;
                    NEWONE:=MAP.NODE(.CURRENT.).CDPTR;
                    POS(.LEVEL.).CURRENT:=NEWONE;
                    POS(.LEVEL.).NEXTONE:=
                         MAP.NODE(.NEWONE.).SPPTR;
                    POS(.LEVEL.).LASTONE:=CURRENT;
                    END;
               END;
          ACROSS:BEGIN
               IF (NEXTONE<=0) THEN
                    STATUS:=ENDLEV
               ELSE BEGIN
                    LASTONE:=CURRENT;
                    CURRENT:=NEXTONE;
                    NEXTONE:=MAP.NODE(.CURRENT.).SPPTR;
                    END;
               END;
          END;
     END;
END;   (* OF AMOVE *)
```

```
PROCEDURE AMAPTRAV(VAR MAP:ADAMMAP;VAR POS:MAPPOS;
     VAR LEVEL:INTEGER;OPER:MAPOPER;VAR RESULT:TRAVRES;
     VAR PRFILE:PFILE);
     (* TRAVERSES THE ADAM MAP ONE NODE AT A TIME IN
     THE DIRECTIONS INDICATED BY OPER.  KEEPS TRACK OF THE
     POSITION IN POS AND LEVEL, AND RETURNS THE RESULTING NODE
     TYPE AS FOLLOWS:
               TOP --- DONE, ERROR OR NOT
               LEAF--- REACHED A LEAF
               NEWNODE--- NEW NODE- OLD LEVEL, OR
                          NEW NODE- NEW LEVEL
               OLDNODE--- OLD NODE- OLD LEVEL  *)
     (* THIS ROUTINE IS A STATE MACHINE.
               STATE     INPUT     NEXT STATE     RESULT
               UP        ATTOP     STOP           TOP
                         OTHERS    STOP           NEWNODE
               DOWN      EMPTY     STOP           TOP
                         ATBOT     STOP           TOP
                         TERM      STOP           LEAF
                         OTHERS    STOP           NEWNODE
               ACROSS    ENDLEV    UP             ----
                         OTHERS    STOP           NEWNODE
               STOP      ----      ----           ----
               *)
VAR  CUROP:MAPOPER;
     STATUS:MAPSTAT;
BEGIN
CUROP:=OPER;
WHILE (CUROP<>STOP) DO
     BEGIN      (* PERFORM ONE OPERATION PER LOOP. *)
     CASE CUROP OF
          DOWN:BEGIN
               CUROP:=STOP;
               AMOVE(MAP,POS,LEVEL,DOWN,STATUS);
               IF (STATUS=EMPTY) THEN
                    BEGIN
                    WRITELN(PRFILE,' *** MAP ERROR.  MISSING ',
                         'SUBTREE. ***');
                    RESULT:=TOP;
                    END
               ELSE IF (STATUS=ATBOT) THEN
                    BEGIN
                    WRITELN(PRFILE,' *** MAP ERROR.  MISSING ',
                         'LEAF. ***');
                    RESULT:=TOP;
                    END
               ELSE IF (STATUS=TERM) THEN
                    RESULT:=LEAF
               ELSE   (* NEW LEVEL. *)
                    RESULT:=NEWNODE;
               END;
          ACROSS:BEGIN
```

```
                              AMOVE(MAP,POS,LEVEL,ACROSS,STATUS);
                              IF (STATUS=ENDLEV) THEN CUROP:=UP
                              ELSE BEGIN
                                   CUROP:=STOP;
                                   RESULT:=NEWNODE;
                                   END;
                              END;
                        UP:  BEGIN
                              CUROP:=STOP;
                              AMOVE(MAP,POS,LEVEL,UP,STATUS);
                              IF (STATUS=ATTOP) THEN RESULT:=TOP
                              ELSE RESULT:=OLDNODE;
                              END;
                        END;
                  END;
            END;   (* OF AMAPTRAV *)

            BEGIN           (* ADAMTEST *)
            (*$NULLBODY*)
            END.
```

```
PROGRAM ADAMUTIL;
     (* THIS PROGRAM TESTS THE ADAM PROCEDURES *)

CONST
     MAXNODES = 50;     (* SMALL BUFFER FOR NOW. *)
     MAXDIM = 10;       (* MAX NUMBER OF DIMENSIONS. *)
     MAXBIT = 16;       (* INTEGER LENGTH IN BITS. *)
          (* MAXBIT SHOULD ALWAYS BE BIGGER THAN MAXDIM. *)
     MAXDI = 100;        (* MAX DATA ITEMS IN A RETRIEVAL. *)

TYPE NODECELL = RECORD
          SPPTR,CDPTR:INTEGER;
          DATA:ARRAY (. 1..2 .) OF INTEGER;
          END;
     NODEARRAY = ARRAY (. 0..MAXNODES .) OF NODECELL;
     MAPDIRECT = RECORD
          NUMDIM,NUMREG,NUMNODES,NUMLEV:INTEGER;
          END;
     ADAMMAP = RECORD
          DIRECT:MAPDIRECT;
          NODE:NODEARRAY;
          END;
     PFILE = TEXT;
     BITSET = ARRAY (. 1..MAXBIT .) OF BOOLEAN;
     BITARRAY = ARRAY (. 1..MAXBIT .) OF BITSET;
     DIVEC = ARRAY (. 0..MAXBIT .) OF INTEGER;
     PUFLAG = (PACK,UNPACK);
     SCHRES = (NOTDONE,INSERT,MATCH);
     REGDEF = RECORD
          LOWVAL,HIGHVAL:DIVEC;
          END;
     SEQDS = ARRAY (. 0..MAXDI .) OF DIVEC;
     C8ARAY = ARRAY (. 1..8 .) OF CHAR;

VAR  MAP:ADAMMAP;
     MASK:ARRAY (. 1..MAXBIT .) OF INTEGER;
     PRFILE:PFILE;
     PRFLAG:(DISPLAY,PRINTER);
     BITFLG,VECFLG:BOOLEAN;         (* DEBUG FLAGS. *)

     (*        I/O ROUTINES.     *)

PROCEDURE OUTBITS(OFILE:PFILE;BITS:BITSET;NUM:INTEGER);
     EXTERNAL;
     (* THIS PROCEDURE OUTPUTS NUM BITS TO OFILE. *)

PROCEDURE OUTVEC(DATVEC:DIVEC);EXTERNAL;
     (* OUTPUTS DATVEC TO PRFILE IN HEX AND INTEGER. *)
(*                    FREE NODE MANIPULATION ROUTINES              *)

PROCEDURE GETCELL(VAR NODE:NODEARRAY;VAR NEW:INTEGER);
     (* RETURNS THE INDEX OF THE NEXT FREE NODE AFTER
     REMOVING IT FROM THE FREE LIST.
                 NEW=0 ON RETURN
                       MEANS "OUT OF FREE STORAGE."
*)
BEGIN
NEW:=NODE(. 0 .).SPPTR;        (* POINTER TO FIRST FREE CELL. *)
NODE(. 0 .).SPPTR:=NODE(. NEW .).SPPTR; (* NEXT CELL. *)
IF NEW=NODE(. 0 .).CDPTR THEN
      BEGIN                    (* BUFFER EMPTY. *)
      NODE(. 0 .).CDPTR:=0;
      WRITELN('  *** MAP FULL. ***');
      END;
END;   (* OF GETCELL *)
```

```
PROCEDURE NEWBUFF(VAR NODE:NODEARRAY;M:INTEGER);
     (* INITIALIZES A DATA BUFFER OF LENGTH M IN THE NODE
     ARRAY INTO A FREE STORAGE LIST.
          THIS SETS UP NODE(0) AS THE HEAD OF THE LIST
     OF FREE NODES.
               NODE(0).SPPTR-----POINTER TO NEXT FREE NODE.
               NODE(0).CDPTR-----POINTER TO LAST FREE NODE.
               NODE(0).DATA(1)---NUMBER OF NODES.
*)
VAR  I,J:INTEGER;

BEGIN
I:=0;
J:=1;     (* J ALWAYS LEADS I BY ONE. *)
WHILE (I<=M) AND (I<=MAXNODES) DO
     BEGIN               (* LOOP THROUGH ALL NODES. *)
     WITH NODE(. I .) DO
          BEGIN          (* SET UP EACH NODE'S FIELDS. *)
          SPPTR:=J;      (* POINTS TO NEXT NODE. *)
          CDPTR:=0;      (* NULL CHILD POINTER. *)
          DATA(. 1 .):=0;(* EMPTY KEY FIELD. *)
          DATA(. 2 .):=0;(* NO REGIONS. *)
          END;
     I:=J;               (* ADVANCE TO NEXT NODE. *)
     J:=J+1;
     END;
I:=I-1;                  (* RETURN TO LAST NODE. *)
NODE(. 0 .).CDPTR:=I;    (* POINT TO LAST NODE. *)
NODE(. 0 .).DATA(. 1 .):=I; (* NUMBER OF NODES. *)
NODE(. I .).SPPTR:=0;    (* LAST NODE IN FREE LIST. *)
END;   (* OF NEWBUFF *)
```

```
PROCEDURE RETCELL(VAR NODE:NODEARRAY;OLD:INTEGER);
     (* RETURNS AN OLD NODE TO THE FREE STORAGE LIST
     AFTER CHECKING FOR DETECTABLE ERRORS. *)
BEGIN
IF (0<OLD) AND (OLD<=NODE(. 0 .).DATA(. 1 .)) THEN
     BEGIN               (* ASSUME OLD IS VALID. *)
     WITH NODE(. OLD .) DO
          BEGIN
          SPPTR:=NODE(. 0 .).SPPTR; (* NEXT CELL *)
          CDPTR:=0;      (* ZERO REST OF NODE. *)
          DATA(. 1 .):=0;
          DATA(. 2 .):=0;
          END;
     WITH NODE(. 0 .) DO
          BEGIN
          SPPTR:=OLD; (* INSERT AT HEAD OF LIST. *)
          IF (CDPTR=0) THEN
               CDPTR:=OLD; (* LAST CELL ON FREE LIST. *)
          END;
     END;
END;   (* OF RETCELL *)
```

```
(*              BIT MANIPULATION UTILITIES.            *)

PROCEDURE BITPU(VAR BITS:BITSET;VAR INTWD:INTEGER;DIR:PUFLAG);
     (* KEYS ON DIR (PACK OR UNPACK) TO
               PACK BITS INTO INTWD, OR
               UNPACK INTWD INTO BITS.  *)
VAR  INDEX,REVIND,BITWD:INTEGER;

BEGIN
IF (DIR=UNPACK) THEN
     BEGIN
     BITWD:=INTWD;
     BITS(.1.):=(BITWD<0);          (* SIGN BIT. *)
     IF (BITS(.1.) ) THEN BITWD:=BITWD+MAXINT+1;
               (* USE JUST BITWD+MAXINT ON ONE'S COMPLEMENT
               MACHINE. THIS IS FOR TWO'S COMPLEMENT ONLY. *)
     END
ELSE
     BITWD:=0;             (* BUILD WORD HERE. *)
INDEX:=2;
WHILE (INDEX<=MAXBIT) DO
     BEGIN                 (* CHECK THE BITS ONE AT A TIME. *)
     REVIND:=MAXBIT-INDEX+1;
     IF (DIR=UNPACK) THEN
          BEGIN            (* GET NEXT BIT SETTING. *)
          BITS(.INDEX.):=(BITWD>=MASK(.REVIND.));
          IF (BITS(.INDEX.)) THEN    (* UNPACK THE BIT. *)
               BITWD:=BITWD-MASK(.REVIND.);
          END
     ELSE
          IF (BITS(.INDEX.)) THEN    (* PACK ON THE BIT. *)
               BITWD:=BITWD+MASK(.REVIND.);
     INDEX:=INDEX+1;
     END;
IF (DIR = PACK) THEN
     BEGIN                 (* CHECK THE SIGN BIT. *)
     IF (BITS(.1.)) THEN
          BITWD:=BITWD-MAXINT-1;   (*TWO'S COMPLEMENT MACHINE
               INVERSE OF ABOVE TRANSFORMATION. *)
     INTWD:=BITWD;
     END;
END;   (* OF BITPU *)
```

```
PROCEDURE CRTLSK(VAR LSKVEC:DIVEC;DATVEC:DIVEC;K:INTEGER);
     (* CREATE THE LEVEL SEARCH KEYS FROM THE DATA ITEM VECTOR,
     DATVEC.  THE ITH LEVEL SEARCH KEY CONSISTS OF THE ITH MSB
     OF EACH OF THE ELEMENTS OF DATVEC.  THERE ARE K ELEMENTS
     IN DATVEC, AND MAXBIT ELEMENTS IN LSKVEC. *)
VAR  I,J,FLSEWD,REVIND:INTEGER;
     TBIT:BOOLEAN;
     TBVEC:BITSET;
     LSKT:BITARRAY;
BEGIN
FLSEWD:=0;
I:=1;
WHILE (I<=MAXBIT) DO
     BEGIN                (* ONE VECTOR ELEMENT INTO ONE ROW
                          IN THE BIT ARRAY. *)
     REVIND:=MAXBIT-I+1;
     IF (BITFLG) THEN WRITE(PRFILE,I:3,'  ');
     IF (I<=K) THEN       (* TRANSLATE A ROW. *)
          BITPU(LSKT(.REVIND.),DATVEC(. I .),UNPACK)
     ELSE                 (* SET ROW TO "0"S. *)
          BITPU(LSKT(.REVIND.),FLSEWD,UNPACK);
     IF (BITFLG) THEN OUTBITS(PRFILE,LSKT(.REVIND.),MAXBIT);
     I:=I+1;
     END;
IF (BITFLG) THEN WRITELN(PRFILE);
I:=1;
WHILE (I<=MAXBIT) DO
     BEGIN                (* TRANSPOSE THE BIT ARRAY. *)
     IF (BITFLG) THEN WRITE(PRFILE,I:3,'  ');
     J:=I+1;
     WHILE(J<=MAXBIT) DO
          BEGIN  (* SWITCH ELEMENTS IN THE REST OF THE
                 ROW AND COLUMN. *)
          TBIT:=LSKT(. I,J .);
          LSKT(. I,J .):=LSKT(. J,I .);
          LSKT(. J,I .):=TBIT;
          J:=J+1;
          END;
     IF (BITFLG) THEN OUTBITS(PRFILE,LSKT(.I.),MAXBIT);
     BITPU(LSKT(. I .),LSKVEC(.I.),PACK);
     I:=I+1;
     END;
END;   (* OF CRTLSK *)
```

```
(*              MAP MANIPULATION UTILITIES.            *)

PROCEDURE NODEINS(VAR MAP:ADAMMAP;NEWONE,PARENT,LASTONE,
     NEXTONE:INTEGER);
     (* INSERTS THE NODE AT NEWONE BETWEEN LASTONE AND NEXTONE,
     COMPENSATING IF EITHER IS A PARENT. *)
BEGIN
WITH MAP DO
     BEGIN
     IF (0<NEWONE) AND (NEWONE<=DIRECT.NUMNODES) THEN
          BEGIN          (* NEWONE APPEARS LEGAL. *)
          IF (LASTONE=PARENT) THEN
               BEGIN     (* INSERT BELOW PARENT. *)
               IF (NODE(. PARENT .).CDPTR=0) THEN
                    NEXTONE:=-PARENT; (* NEW LEVEL. *)
               NODE(. NEWONE .).SPPTR:=NEXTONE;
               NODE(. PARENT .).CDPTR:=NEWONE;
               END
          ELSE           (* INSERT INTO LEVEL. *)
               BEGIN
               NODE(. LASTONE .).SPPTR:=NEWONE;
               NODE(. NEWONE .).SPPTR:=NEXTONE;
               END;
          END;
     END;                     (* IGNORE BAD POINTERS. *)
END;   (* OF NODEINS *)
```

```
PROCEDURE MAPSRCH(VAR LSEARCH:SCHRES;VAR PARENT,LASTONE,
     NEXTONE,LEVEL:INTEGER;VAR MAP:ADAMMAP;
     VAR LSK:DIVEC);
     (* SEARCHES THE MAP IN MAP FOR THE LEVEL SEARCH
     KEYS IN LSK.  LSEARCH RETURNS INSERT OR MATCH, AND THE
     POSITION IS RETURNED IN LEVEL, PARENT, LASTONE, AND
     NEXTONE. *)
VAR  L:INTEGER;
BEGIN
NEXTONE:=1;              (* START AT ROOT. *)
LEVEL:=1;
PARENT:=1;
LASTONE:=1;
WITH MAP DO
     BEGIN
     L:=DIRECT.NUMLEV;
     LSEARCH:=MATCH;
     WHILE (LSEARCH<>INSERT) AND (LEVEL<L) DO
          BEGIN                (* START A NEW LEVEL. *)
          PARENT:=NEXTONE;
          LASTONE:=NEXTONE;
          NEXTONE:=NODE(. LASTONE .).CDPTR;  (* MOVE DOWN. *)
          LEVEL:=LEVEL+1;
          LSEARCH:=NOTDONE;
          WHILE (LSEARCH=NOTDONE) DO
               BEGIN           (* CHECK NEXT NODE ON LEVEL. *)
               IF (NEXTONE<=0) THEN
                    LSEARCH:=INSERT (* END OF LEVEL, INSERT. *)
               ELSE
                    BEGIN            (* NOT END, CHECK NODE. *)
                    IF (LSK(.LEVEL.) >
                         NODE(.NEXTONE.).DATA(. 1 .)) THEN
                         BEGIN  (* MOVE ACROSS LEVEL. *)
                         LASTONE:=NEXTONE;
                         NEXTONE:=NODE(.NEXTONE.).SPPTR;
                         END
                    ELSE IF (LSK(.LEVEL.) =
                         NODE(.NEXTONE.).DATA(. 1 .)) THEN
                         LSEARCH:=MATCH  (* GO TO NEXT
                                             LEVEL. *)
                    ELSE LSEARCH:=INSERT; (* INSERT HERE. *)
                    END;
               END;
          END;
     END;
END;  (* OF MAPSRCH *)

BEGIN           (* ADAMTEST *)
(*$NULLBODY*)
END.
```

APPENDIX E

TEST RUNS

Test Run List

```
LDOS READY
ADAM
INPUT    =

OUTPUT   =

PRFILE   = :L

 ADAM TEST ROUTINE.
 ADD  - - - - - - - ADD DATA TO THE MAP.
 DEB  - - - - - - - DEBUG THE STRUCTURE.
 DEL  - - - - - - - DELETE ALL DATA POINTS WITHIN A
                    REGION.
 FIND - - - - - - - FLAG THE NEEDED NODES TO DEFINE
                    A REGION
 GET  - - - - - - - RETRIEVE A REGION OF DATA POINTS
                    INTO A SEQUENTIAL DATA FORM.
 HELP - - - - - - - PRINT OUT HELP TABLE.
 NEW  - - - - - - - CREATE A NEW MAP BUFFER.
 STOP - - - - - - - STOP PROCESSING.
```

```
ADAM COMMAND= NEW
INPUT BUF SIZE >50
INPUT NUM DIM  >3
INPUT NUM LEV  >6
ADAM COMMAND= DEB
 DEBUG COMMAND = HELP
 BITS - - - - SWITCHES BIT OUTPUT FLAG.
 DIR  - - - - OUTPUT MAP DIRECTORY.
 DISP - - - - SWITCHES OUTPUT TO DISPLAY.
 DUMP - - - - PRINTS THE MAP.
 HELP - - - - DISPLAYS THIS TABLE.
 LIST - - - - SWITCHES OUTPUT TO PRINTER.
 PAGE - - - - OUTPUTS A PAGE ON PRINT FILE.
 STOP - - - - EXIT DEBUG MODE.
 VECS - - - - SWITCHES VECTOR OUTPUT FLAG.
 DEBUG COMMAND = LIST
 DEBUG COMMAND = DIR
MAP DIRECTORY.
 SIZE =      50  NODES.
         0 REGIONS
         3 DIMENSIONS
         6 LEVELS
 DEBUG COMMAND = DUMP
 DUMP NODE CONTENTS.
   INDEX OF STARTING NODE = 0
   INDEX OF LAST NODE = 50
DUMP OF ADAM MAP.
    FROM      0     TO     50
     0      S/P=      2  C/D=      50  DATA=  0032  0000
     1      S/P=     -1  C/D=       0  DATA=  0000  0000
     2      S/P=      3  C/D=       0  DATA=  0000  0000
     3      S/P=      4  C/D=       0  DATA=  0000  0000
     4      S/P=      5  C/D=       0  DATA=  0000  0000
     5      S/P=      6  C/D=       0  DATA=  0000  0000
     6      S/P=      7  C/D=       0  DATA=  0000  0000
     7      S/P=      8  C/D=       0  DATA=  0000  0000
     8      S/P=      9  C/D=       0  DATA=  0000  0000
     9      S/P=     10  C/D=       0  DATA=  0000  0000
    10      S/P=     11  C/D=       0  DATA=  0000  0000
    11      S/P=     12  C/D=       0  DATA=  0000  0000
    12      S/P=     13  C/D=       0  DATA=  0000  0000
    13      S/P=     14  C/D=       0  DATA=  0000  0000
    14      S/P=     15  C/D=       0  DATA=  0000  0000
    15      S/P=     16  C/D=       0  DATA=  0000  0000
    16      S/P=     17  C/D=       0  DATA=  0000  0000
    17      S/P=     18  C/D=       0  DATA=  0000  0000
    18      S/P=     19  C/D=       0  DATA=  0000  0000
    19      S/P=     20  C/D=       0  DATA=  0000  0000
    20      S/P=     21  C/D=       0  DATA=  0000  0000
    21      S/P=     22  C/D=       0  DATA=  0000  0000
    22      S/P=     23  C/D=       0  DATA=  0000  0000
    23      S/P=     24  C/D=       0  DATA=  0000  0000
    24      S/P=     25  C/D=       0  DATA=  0000  0000
    25      S/P=     26  C/D=       0  DATA=  0000  0000
```

```
     26      S/P=    27  C/D=      0  DATA=  0000  0000
     27      S/P=    28  C/D=      0  DATA=  0000  0000
     28      S/P=    29  C/D=      0  DATA=  0000  0000
     29      S/P=    30  C/D=      0  DATA=  0000  0000
     30      S/P=    31  C/D=      0  DATA=  0000  0000
     31      S/P=    32  C/D=      0  DATA=  0000  0000
     32      S/P=    33  C/D=      0  DATA=  0000  0000
     33      S/P=    34  C/D=      0  DATA=  0000  0000
     34      S/P=    35  C/D=      0  DATA=  0000  0000
     35      S/P=    36  C/D=      0  DATA=  0000  0000
     36      S/P=    37  C/D=      0  DATA=  0000  0000
     37      S/P=    38  C/D=      0  DATA=  0000  0000
     38      S/P=    39  C/D=      0  DATA=  0000  0000
     39      S/P=    40  C/D=      0  DATA=  0000  0000
     40      S/P=    41  C/D=      0  DATA=  0000  0000
     41      S/P=    42  C/D=      0  DATA=  0000  0000
     42      S/P=    43  C/D=      0  DATA=  0000  0000
     43      S/P=    44  C/D=      0  DATA=  0000  0000
     44      S/P=    45  C/D=      0  DATA=  0000  0000
     45      S/P=    46  C/D=      0  DATA=  0000  0000
     46      S/P=    47  C/D=      0  DATA=  0000  0000
     47      S/P=    48  C/D=      0  DATA=  0000  0000
     48      S/P=    49  C/D=      0  DATA=  0000  0000
     49      S/P=    50  C/D=      0  DATA=  0000  0000
     50      S/P=     0  C/D=      0  DATA=  0000  0000

DEBUG COMMAND = STOP
STOPPING
END OF DEBUG.
```

```
ADAM COMMAND= ADD
VECTOR INPUT FOR DATA PT
 INPUT THE  3 DIMENSIONAL DATA ITEM VECTOR.
 USE A VALUE <0 TO ABORT INPUT SEQUENCE.
    ALL INPUTS SHOULD BE
        0<= X(I) <1
   X( 1) = .35
   X( 2) = .45
   X( 3) = .45
INPUT PT INDEX >1
ADAM COMMAND= ADD
VECTOR INPUT FOR DATA PT
 INPUT THE  3 DIMENSIONAL DATA ITEM VECTOR.
 USE A VALUE <0 TO ABORT INPUT SEQUENCE.
    ALL INPUTS SHOULD BE
        0<= X(I) <1
   X( 1) = .55
   X( 2) = .35
   X( 3) = .45
INPUT PT INDEX >2
ADAM COMMAND= DEB
 DEBUG COMMAND = PAGE
 DEBUG COMMAND = DUMP
 DUMP NODE CONTENTS.
   INDEX OF STARTING NODE = 0
   INDEX OF LAST NODE = 50
DUMP OF ADAM MAP.
    FROM      0     TO    50
     0      S/P=      12  C/D=      50  DATA=  0032  0000
     1      S/P=      -1  C/D=       2  DATA=  0000  0000
     2      S/P=       7  C/D=       3  DATA=  0000  0000
     3      S/P=      -2  C/D=       4  DATA=  0007  0000
     4      S/P=      -3  C/D=       5  DATA=  0006  0000
     5      S/P=      -4  C/D=       6  DATA=  0007  0000
     6      S/P=      -5  C/D=      -1  DATA=  0001  0000
     7      S/P=      -1  C/D=       8  DATA=  0001  0000
     8      S/P=      -7  C/D=       9  DATA=  0006  0000
     9      S/P=      -8  C/D=      10  DATA=  0004  0000
    10      S/P=      -9  C/D=      11  DATA=  0006  0000
    11      S/P=     -10  C/D=      -2  DATA=  0003  0000
    12      S/P=      13  C/D=       0  DATA=  0000  0000
    13      S/P=      14  C/D=       0  DATA=  0000  0000
    14      S/P=      15  C/D=       0  DATA=  0000  0000
    15      S/P=      16  C/D=       0  DATA=  0000  0000
    16      S/P=      17  C/D=       0  DATA=  0000  0000
    17      S/P=      18  C/D=       0  DATA=  0000  0000
    18      S/P=      19  C/D=       0  DATA=  0000  0000
    19      S/P=      20  C/D=       0  DATA=  0000  0000
    20      S/P=      21  C/D=       0  DATA=  0000  0000
    21      S/P=      22  C/D=       0  DATA=  0000  0000
    22      S/P=      23  C/D=       0  DATA=  0000  0000
    23      S/P=      24  C/D=       0  DATA=  0000  0000
    24      S/P=      25  C/D=       0  DATA=  0000  0000
    25      S/P=      26  C/D=       0  DATA=  0000  0000
    26        S/P=     27  C/D=       0  DATA=  0000  0000
    27        S/P=     28  C/D=       0  DATA=  0000  0000
    28        S/P=     29  C/D=       0  DATA=  0000  0000
    29        S/P=     30  C/D=       0  DATA=  0000  0000
    30        S/P=     31  C/D=       0  DATA=  0000  0000
    31        S/P=     32  C/D=       0  DATA=  0000  0000
    32        S/P=     33  C/D=       0  DATA=  0000  0000
    33        S/P=     34  C/D=       0  DATA=  0000  0000
    34        S/P=     35  C/D=       0  DATA=  0000  0000
    35        S/P=     36  C/D=       0  DATA=  0000  0000
    36        S/P=     37  C/D=       0  DATA=  0000  0000
    37        S/P=     38  C/D=       0  DATA=  0000  0000
    38        S/P=     39  C/D=       0  DATA=  0000  0000
    39        S/P=     40  C/D=       0  DATA=  0000  0000
    40        S/P=     41  C/D=       0  DATA=  0000  0000
    41        S/P=     42  C/D=       0  DATA=  0000  0000
    42        S/P=     43  C/D=       0  DATA=  0000  0000
    43        S/P=     44  C/D=       0  DATA=  0000  0000
    44        S/P=     45  C/D=       0  DATA=  0000  0000
    45        S/P=     46  C/D=       0  DATA=  0000  0000
    46        S/P=     47  C/D=       0  DATA=  0000  0000
    47        S/P=     48  C/D=       0  DATA=  0000  0000
    48        S/P=     49  C/D=       0  DATA=  0000  0000
    49        S/P=     50  C/D=       0  DATA=  0000  0000
    50        S/P=      0  C/D=       0  DATA=  0000  0000

 DEBUG COMMAND = STOP
 STOPPING
 END OF DEBUG.
ADAM COMMAND= ADD
VECTOR INPUT FOR DATA PT
 INPUT THE  3 DIMENSIONAL DATA ITEM VECTOR.
 USE A VALUE <0 TO ABORT INPUT SEQUENCE.
    ALL INPUTS SHOULD BE
        0<= X(I) <1
   X( 1) = .45
   X( 2) = .35
   X( 3) = .35
INPUT PT INDEX >3
ADAM COMMAND= ADD
VECTOR INPUT FOR DATA PT
 INPUT THE  3 DIMENSIONAL DATA ITEM VECTOR.
 USE A VALUE <0 TO ABORT INPUT SEQUENCE.
    ALL INPUTS SHOULD BE
        0<= X(I) <1
   X( 1) = .55
   X( 2) = .45
   X( 3) = .35
INPUT PT INDEX >4
ADAM COMMAND= ADD
VECTOR INPUT FOR DATA PT
 INPUT THE  3 DIMENSIONAL DATA ITEM VECTOR.
 USE A VALUE <0 TO ABORT INPUT SEQUENCE.
    ALL INPUTS SHOULD BE
```

```
          0<= X(I) <1
     X( 1) = .45
     X( 2) = .25
     X( 3) = .25
INPUT PT INDEX >5
ADAM COMMAND= ADD
VECTOR INPUT FOR DATA PT
 INPUT THE  3 DIMENSIONAL DATA ITEM VECTOR.
 USE A VALUE <0 TO ABORT INPUT SEQUENCE.
      ALL INPUTS SHOULD BE
          0<= X(I) <1
     X( 1) = .35
     X( 2) = .55
     X( 3) = .15
INPUT PT INDEX >6
ADAM COMMAND= ADD
VECTOR INPUT FOR DATA PT
 INPUT THE  3 DIMENSIONAL DATA ITEM VECTOR.
 USE A VALUE <0 TO ABORT INPUT SEQUENCE.
      ALL INPUTS SHOULD BE
          0<= X(I) <1
     X( 1) = 0
     X( 2) = 0
     X( 3) = 0
INPUT PT INDEX >7
ADAM COMMAND= ADD
VECTOR INPUT FOR DATA PT
 INPUT THE  3 DIMENSIONAL DATA ITEM VECTOR.
 USE A VALUE <0 TO ABORT INPUT SEQUENCE.
      ALL INPUTS SHOULD BE
          0<= X(I) <1
     X( 1) = .1
     X( 2) = .1
     X( 3) = .1
INPUT PT INDEX >8
ADAM COMMAND= ADD
VECTOR INPUT FOR DATA PT
 INPUT THE  3 DIMENSIONAL DATA ITEM VECTOR.
 USE A VALUE <0 TO ABORT INPUT SEQUENCE.
      ALL INPUTS SHOULD BE
          0<= X(I) <1
     X( 1) = .9
     X( 2) = .9
     X( 3) = .9
INPUT PT INDEX >9
ADAM COMMAND= ADD
VECTOR INPUT FOR DATA PT
 INPUT THE  3 DIMENSIONAL DATA ITEM VECTOR.
 USE A VALUE <0 TO ABORT INPUT SEQUENCE.
      ALL INPUTS SHOULD BE
          0<= X(I) <1
     X( 1) = 0
     X( 2) = .5
     X( 3) = .9
```

```
INPUT PT INDEX >10
ADAM COMMAND= ADD
VECTOR INPUT FOR DATA PT
 INPUT THE  3 DIMENSIONAL DATA ITEM VECTOR.
 USE A VALUE <0 TO ABORT INPUT SEQUENCE.
    ALL INPUTS SHOULD BE
        0<= X(I) <1
   X( 1) = .9
   X( 2) = 0
   X( 3) = .5
INPUT PT INDEX >11
ADAM COMMAND= ADD
VECTOR INPUT FOR DATA PT
 INPUT THE  3 DIMENSIONAL DATA ITEM VECTOR.
 USE A VALUE <0 TO ABORT INPUT SEQUENCE.
    ALL INPUTS SHOULD BE
        0<= X(I) <1
   X( 1) = .5
   X( 2) = .5
   X( 3) = .5
INPUT PT INDEX >12
ADAM COMMAND= ADD
VECTOR INPUT FOR DATA PT
 INPUT THE  3 DIMENSIONAL DATA ITEM VECTOR.
 USE A VALUE <0 TO ABORT INPUT SEQUENCE.
    ALL INPUTS SHOULD BE
        0<= X(I) <1
   X( 1) = .75
   X( 2) = .25
   X( 3) = .5
INPUT PT INDEX >13
 *** MAP FULL. ***
 *** MAP FULL. ***
ADAM COMMAND= DEB
 DEBUG COMMAND = PAGE
 DEBUG COMMAND = DUMP
 DUMP NODE CONTENTS.
   INDEX OF STARTING NODE = 0
   INDEX OF LAST NODE = 50
DUMP OF ADAM MAP.
    FROM      0     TO    50
     0      S/P=     50  C/D=     50  DATA=  0032  0000
     1      S/P=     -1  C/D=      2  DATA=  0000  0000
     2      S/P=      7  C/D=     25  DATA=  0000  0000
     3      S/P=     -2  C/D=     12  DATA=  0007  0000
     4      S/P=     -3  C/D=      5  DATA=  0006  0000
     5      S/P=     -4  C/D=      6  DATA=  0007  0000
     6      S/P=     -5  C/D=     -1  DATA=  0001  0000
     7      S/P=     20  C/D=      8  DATA=  0001  0000
     8      S/P=     -7  C/D=     15  DATA=  0006  0000
     9      S/P=     -8  C/D=     10  DATA=  0004  0000
    10      S/P=     -9  C/D=     11  DATA=  0006  0000
    11      S/P=    -10  C/D=     -2  DATA=  0003  0000
    12      S/P=      4  C/D=     18  DATA=  0001  0000
   13         S/P=     -12  C/D=      14  DATA=  0007   0000
   14         S/P=     -13  C/D=      -3  DATA=  0006   0000
   15         S/P=       9  C/D=      16  DATA=  0002   0000
   16         S/P=     -15  C/D=      17  DATA=  0006   0000
   17         S/P=     -16  C/D=      -4  DATA=  0005   0000
   18         S/P=      13  C/D=      19  DATA=  0001   0000
   19         S/P=     -18  C/D=      -5  DATA=  0000   0000
   20         S/P=      41  C/D=      21  DATA=  0002   0000
   21         S/P=     -20  C/D=      22  DATA=  0001   0000
   22         S/P=     -21  C/D=      23  DATA=  0004   0000
   23         S/P=     -22  C/D=      24  DATA=  0001   0000
   24         S/P=     -23  C/D=      -6  DATA=  0003   0000
   25         S/P=       3  C/D=      26  DATA=  0000   0000
   26         S/P=     -25  C/D=      27  DATA=  0000   0000
   27         S/P=      29  C/D=      28  DATA=  0000   0000
   28         S/P=     -27  C/D=      -7  DATA=  0000   0000
   29         S/P=     -26  C/D=      30  DATA=  0007   0000
   30         S/P=     -29  C/D=      -8  DATA=  0007   0000
   31         S/P=      -1  C/D=      46  DATA=  0007   0000
   32         S/P=     -31  C/D=      33  DATA=  0007   0000
   33         S/P=     -32  C/D=      34  DATA=  0007   0000
   34         S/P=     -33  C/D=      35  DATA=  0000   0000
   35         S/P=     -34  C/D=      -9  DATA=  0000   0000
   36         S/P=      31  C/D=      37  DATA=  0006   0000
   37         S/P=     -36  C/D=      38  DATA=  0004   0000
   38         S/P=     -37  C/D=      39  DATA=  0004   0000
   39         S/P=     -38  C/D=      40  DATA=  0000   0000
   40         S/P=     -39  C/D=     -10  DATA=  0000   0000
   41         S/P=      36  C/D=      42  DATA=  0005   0000
   42         S/P=     -41  C/D=      43  DATA=  0001   0000
   43         S/P=     -42  C/D=      44  DATA=  0001   0000
   44         S/P=     -43  C/D=      45  DATA=  0000   0000
   45         S/P=     -44  C/D=     -11  DATA=  0000   0000
   46         S/P=      32  C/D=      47  DATA=  0000   0000
   47         S/P=     -46  C/D=      48  DATA=  0000   0000
   48         S/P=     -47  C/D=      49  DATA=  0000   0000
   49         S/P=     -48  C/D=     -12  DATA=  0000   0000
   50         S/P=       0  C/D=       0  DATA=  0000   0000

DEBUG COMMAND = STOP
STOPPING
END OF DEBUG.
```

```
ADAM COMMAND= GET
INPUT REG IND  >0
RETRIEVED DATA POINTS.
      7            .00000          .00000          .00000
      8      .09375     .09375     .09375
      5      .43750     .25000     .25000
      3      .43750     .34375     .34375
      1      .34375     .43750     .43750
      4      .53125     .43750     .34375
      2      .53125     .34375     .43750
      6      .34375     .53125     .12500
     11      .87500          .00000     .50000
     10            .00000     .50000     .87500
     12      .50000     .50000     .50000
      9      .87500     .87500     .87500
```

```
ADAM COMMAND= FIND
INPUT REG IND  >1
VECTOR INPUT FOR REG MIN
 INPUT THE  3 DIMENSIONAL DATA ITEM VECTOR.
 USE A VALUE <0 TO ABORT INPUT SEQUENCE.
    ALL INPUTS SHOULD BE
        0<= X(I) <1
   X( 1) = .3
   X( 2) = .2
   X( 3) = .1
VECTOR INPUT FOR REG MAX
 INPUT THE  3 DIMENSIONAL DATA ITEM VECTOR.
 USE A VALUE <0 TO ABORT INPUT SEQUENCE.
    ALL INPUTS SHOULD BE
        0<= X(I) <1
   X( 1) = .5
   X( 2) = .6
   X( 3) = .5
ADAM COMMAND= GET
INPUT REG IND  >1
RETRIEVED DATA POINTS.
      5      .43750     .25000     .25000
      3      .43750     .34375     .34375
      1      .34375     .43750     .43750
      6      .34375     .53125     .12500
     12      .50000     .50000     .50000

ADAM COMMAND= DEB
 DEBUG COMMAND = DUMP
 DUMP NODE CONTENTS.
   INDEX OF STARTING NODE = 0
   INDEX OF LAST NODE = 50
DUMP OF ADAM MAP.
    FROM       0      TO     50
     0       S/P=      50  C/D=     50  DATA=  0032  0000
     1       S/P=      -1  C/D=      2  DATA=  0000  C000
     2       S/P=       7  C/D=     25  DATA=  0000  C000
     3       S/P=      -2  C/D=     12  DATA=  0007  C000
     4       S/P=      -3  C/D=      5  DATA=  0006  C000
     5       S/P=      -4  C/D=      6  DATA=  0007  8000
     6       S/P=      -5  C/D=     -1  DATA=  0001  0000
     7       S/P=      20  C/D=      8  DATA=  0001  C000
     8       S/P=      -7  C/D=     15  DATA=  0006  C000
     9       S/P=      -8  C/D=     10  DATA=  0004  C000
    10       S/P=      -9  C/D=     11  DATA=  0006  C000
    11       S/P=     -10  C/D=     -2  DATA=  0003  4000
    12       S/P=       4  C/D=     18  DATA=  0001  8000
    13       S/P=     -12  C/D=     14  DATA=  0007  0000
    14       S/P=     -13  C/D=     -3  DATA=  0006  0000
    15       S/P=       9  C/D=     16  DATA=  0002  C000
    16       S/P=     -15  C/D=     17  DATA=  0006  C000
    17       S/P=     -16  C/D=     -4  DATA=  0005  4000
    18       S/P=      13  C/D=     19  DATA=  0001  0000
```

```
  19       S/P=  .  -18  C/D=     -5  DATA=  0000  0000
  20       S/P=      41  C/D=     21  DATA=  0002  C000
  21       S/P=     -20  C/D=     22  DATA=  0001  C000
  22       S/P=     -21  C/D=     23  DATA=  0004  C000
  23       S/P=     -22  C/D=     24  DATA=  0001  8000
  24       S/P=     -23  C/D=     -6  DATA=  0003  0000
  25       S/P=       3  C/D=     26  DATA=  0000  4000
  26       S/P=     -25  C/D=     27  DATA=  0000  0000
  27       S/P=      29  C/D=     28  DATA=  0000  0000
  28       S/P=     -27  C/D=     -7  DATA=  0000  0000
  29       S/P=     -26  C/D=     30  DATA=  0007  0000
  30       S/P=     -29  C/D=     -8  DATA=  0007  0000
  31       S/P=      -1  C/D=     46  DATA=  0007  C000
  32       S/P=     -31  C/D=     33  DATA=  0007  4000
  33       S/P=     -32  C/D=     34  DATA=  0007  0000
  34       S/P=     -33  C/D=     35  DATA=  0000  0000
  35       S/P=     -34  C/D=     -9  DATA=  0000  0000
  36       S/P=      31  C/D=     37  DATA=  0006  C000
  37       S/P=     -36  C/D=     38  DATA=  0004  4000
  38       S/P=     -37  C/D=     39  DATA=  0004  0000
  39       S/P=     -38  C/D=     40  DATA=  0000  0000
  40       S/P=     -39  C/D=    -10  DATA=  0000  0000
  41       S/P=      36  C/D=     42  DATA=  0005  C000
  42       S/P=     -41  C/D=     43  DATA=  0001  4000
  43       S/P=     -42  C/D=     44  DATA=  0001  0000
  44       S/P=     -43  C/D=     45  DATA=  0000  0000
  45       S/P=     -44  C/D=    -11  DATA=  0000  0000
  46       S/P=      32  C/D=     47  DATA=  0000  C000
  47       S/P=     -46  C/D=     48  DATA=  0000  C000
  48       S/P=     -47  C/D=     49  DATA=  0000  C000
  49       S/P=     -48  C/D=    -12  DATA=  0000  C000
  50       S/P=       0  C/D=      0  DATA=  0000  0000

 DEBUG COMMAND = STOP
 STOPPING
 END OF DEBUG.
ADAM COMMAND= FIND
INPUT REG IND  >2
VECTOR INPUT FOR REG MIN
 INPUT THE  3 DIMENSIONAL DATA ITEM VECTOR.
 USE A VALUE <0 TO ABORT INPUT SEQUENCE.
    ALL INPUTS SHOULD BE
        0<= X(I) <1
   X( 1) = .5
   X( 2) = .3
   X( 3) = .3
VECTOR INPUT FOR REG MAX
 INPUT THE  3 DIMENSIONAL DATA ITEM VECTOR.
 USE A VALUE <0 TO ABORT INPUT SEQUENCE.
    ALL INPUTS SHOULD BE
        0<= X(I) <1
   X( 1) = .6
   X( 2) = .5
   X( 3) = .5
```

```
ADAM COMMAND= GET
INPUT REG IND  >2
RETRIEVED DATA POINTS.
      4       .53125     .43750     .34375
      2       .53125     .34375     .43750
     12       .50000     .50000     .50000

ADAM COMMAND= FIND
INPUT REG IND  >3
VECTOR INPUT FOR REG MIN
 INPUT THE  3 DIMENSIONAL DATA ITEM VECTOR.
 USE A VALUE <0 TO ABORT INPUT SEQUENCE.
    ALL INPUTS SHOULD BE
        0<= X(I) <1
   X( 1) = .3
   X( 2) = .2
   X( 3) = .1
VECTOR INPUT FOR REG MAX
 INPUT THE  3 DIMENSIONAL DATA ITEM VECTOR.
 USE A VALUE <0 TO ABORT INPUT SEQUENCE.
    ALL INPUTS SHOULD BE
        0<= X(I) <1
   X( 1) = .5
   X( 2) = .6
   X( 3) = .5
ADAM COMMAND= GET
INPUT REG IND  >3
RETRIEVED DATA POINTS.
      5       .43750     .25000     .25000
      3       .43750     .34375     .34375
      1       .34375     .43750     .43750
      6       .34375     .53125     .12500
     12       .50000     .50000     .50000

ADAM COMMAND= FIND
INPUT REG IND  >3
VECTOR INPUT FOR REG MIN
 INPUT THE  3 DIMENSIONAL DATA ITEM VECTOR.
 USE A VALUE <0 TO ABORT INPUT SEQUENCE.
    ALL INPUTS SHOULD BE
        0<= X(I) <1
   X( 1) = .4
   X( 2) = .3
   X( 3) = .3
VECTOR INPUT FOR REG MAX
 INPUT THE  3 DIMENSIONAL DATA ITEM VECTOR.
 USE A VALUE <0 TO ABORT INPUT SEQUENCE.
    ALL INPUTS SHOULD BE
        0<= X(I) <1
   X( 1) = .6
   X( 2) = .5
   X( 3) = .5
ADAM COMMAND= GET
INPUT REG IND  >3
```

```
RETRIEVED DATA POINTS.
     3       .43750     .34375     .34375
     1       .34375     .43750     .43750
     4       .53125     .43750     .34375
     2       .53125     .34375     .43750
     6       .34375     .53125     .12500
    12       .50000     .50000     .50000

ADAM COMMAND= DEB
 DEBUG COMMAND = DUMP
 DUMP NODE CONTENTS.
   INDEX OF STARTING NODE = 0
   INDEX OF LAST NODE = 50
DUMP OF ADAM MAP.
    FROM       0      TO     50
      0       S/P=     50  C/D=     50  DATA=  0032  0000
      1       S/P=     -1  C/D=      2  DATA=  0000  FC00
      2       S/P=      7  C/D=     25  DATA=  0000  DC00
      3       S/P=     -2  C/D=     12  DATA=  0007  CC00
      4       S/P=     -3  C/D=      5  DATA=  0006  CC00
      5       S/P=     -4  C/D=      6  DATA=  0007  8800
      6       S/P=     -5  C/D=     -1  DATA=  0001  0000
      7       S/P=     20  C/D=      8  DATA=  0001  FC00
      8       S/P=     -7  C/D=     15  DATA=  0006  FC00
      9       S/P=     -8  C/D=     10  DATA=  0004  FC00
     10       S/P=     -9  C/D=     11  DATA=  0006  E800
     11       S/P=    -10  C/D=     -2  DATA=  0003  4400
     12       S/P=      4  C/D=     18  DATA=  0001  8C00
     13       S/P=    -12  C/D=     14  DATA=  0007  0800
     14       S/P=    -13  C/D=     -3  DATA=  0006  0000
     15       S/P=      9  C/D=     16  DATA=  0002  FC00
     16       S/P=    -15  C/D=     17  DATA=  0006  E800
     17       S/P=    -16  C/D=     -4  DATA=  0005  4400
     18       S/P=     13  C/D=     19  DATA=  0001  0C00
     19       S/P=    -18  C/D=     -5  DATA=  0000  0400
     20       S/P=     41  C/D=     21  DATA=  0002  DC00
     21       S/P=    -20  C/D=     22  DATA=  0001  CC00
     22       S/P=    -21  C/D=     23  DATA=  0004  CC00
     23       S/P=    -22  C/D=     24  DATA=  0001  8800
     24       S/P=    -23  C/D=     -6  DATA=  0003  0000
     25       S/P=      3  C/D=     26  DATA=  0000  4400
     26       S/P=    -25  C/D=     27  DATA=  0000  0000
     27       S/P=     29  C/D=     28  DATA=  0000  0000
     28       S/P=    -27  C/D=     -7  DATA=  0000  0000
     29       S/P=    -26  C/D=     30  DATA=  0007  0000
     30       S/P=    -29  C/D=     -8  DATA=  0007  0000
     31       S/P=     -1  C/D=     46  DATA=  0007  FC00
     32       S/P=    -31  C/D=     33  DATA=  0007  5400
     33       S/P=    -32  C/D=     34  DATA=  0007  0000
     34       S/P=    -33  C/D=     35  DATA=  0000  0000
     35       S/P=    -34  C/D=     -9  DATA=  0000  0000
     36       S/P=     31  C/D=     37  DATA=  0006  DC00
     37       S/P=    -36  C/D=     38  DATA=  0004  4400
     38       S/P=    -37  C/D=     39  DATA=  0004  0000
   39        S/P=     -38  C/D=     40  DATA=  0000   0000
   40        S/P=     -39  C/D=    -10  DATA=  0000   0000
   41        S/P=      36  C/D=     42  DATA=  0005   FC00
   42        S/P=     -41  C/D=     43  DATA=  0001   5400
   43        S/P=     -42  C/D=     44  DATA=  0001   0000
   44        S/P=     -43  C/D=     45  DATA=  0000   0000
   45        S/P=     -44  C/D=    -11  DATA=  0000   0000
   46        S/P=      32  C/D=     47  DATA=  0000   FC00
   47        S/P=     -46  C/D=     48  DATA=  0000   FC00
   48        S/P=     -47  C/D=     49  DATA=  0000   FC00
   49        S/P=     -48  C/D=    -12  DATA=  0000   FC00
   50        S/P=       0  C/D=      0  DATA=  0000   0000

DEBUG COMMAND = STOP
STOPPING
STOPPING
END OF DEBUG.
```

```
ADAM COMMAND= DELETE
ADAM COMMAND= DEL
INPUT REG IND  >2
ADAM COMMAND= FIND
INPUT REG IND  >3
VECTOR INPUT FOR REG MIN
 INPUT THE  3 DIMENSIONAL DATA ITEM VECTOR.
 USE A VALUE <0 TO ABORT INPUT SEQUENCE.
    ALL INPUTS SHOULD BE
        0<= X(I) <1
   X( 1) = -1
ADAM COMMAND= GET
INPUT REG IND  >3
RETRIEVED DATA POINTS.
      3      .46677    .37302    .35934
      1      .37302    .46677    .45309
      6      .37302    .56052    .14059

ADAM COMMAND= GET 0
INPUT REG IND  >0
RETRIEVED DATA POINTS.
      7      .02927    .02927    .01559
      8      .12302    .12302    .10934
      5      .46677    .27927    .26559
      3      .46677    .37302    .35934
      1      .37302    .46677    .45309
      6      .37302    .56052    .14059
     11      .90427    .02927    .51559
     10      .02927    .52927    .89059
      9      .90427    .90427    .89059

ADAM COMMAND= DEB
 DEBUG COMMAND = DUMP
 DUMP NODE CONTENTS.
   INDEX OF STARTING NODE = 0
   INDEX OF LAST NODE = 50
DUMP OF ADAM MAP.
    FROM      0     TO     50
     0      S/P=     46   C/D=     50  DATA=  0032  0000
     1      S/P=     -1   C/D=      2  DATA=  0000  FC00
     2      S/P=     20   C/D=     25  DATA=  0000  DC00
     3      S/P=     -2   C/D=     12  DATA=  0007  CC00
     4      S/P=     -3   C/D=      5  DATA=  0006  CC00
     5      S/P=     -4   C/D=      6  DATA=  0007  8800
     6      S/P=     -5   C/D=     -1  DATA=  0001  0000
     7      S/P=      8   C/D=      0  DATA=  0000  0000
     8      S/P=      9   C/D=      0  DATA=  0000  0000
     9      S/P=     10   C/D=      0  DATA=  0000  0000
    10      S/P=     11   C/D=      0  DATA=  0000  0000
    11      S/P=     15   C/D=      0  DATA=  0000  0000
    12      S/P=      4   C/D=     18  DATA=  0001  8C00
    13      S/P=    -12   C/D=     14  DATA=  0007  0800
    14      S/P=    -13   C/D=     -3  DATA=  0006  0000
    15      S/P=     16   C/D=      0  DATA=  0000  0000
```

```
     16         S/P=      17  C/D=      0  DATA=  0000  0000
     17         S/P=      50  C/D=      0  DATA=  0000  0000
     18         S/P=      13  C/D=     19  DATA=  0001  0C00
     19         S/P=     -18  C/D=     -5  DATA=  0000  0400
     20         S/P=      41  C/D=     21  DATA=  0002  DC00
     21         S/P=     -20  C/D=     22  DATA=  0001  CC00
     22         S/P=     -21  C/D=     23  DATA=  0004  CC00
     23         S/P=     -22  C/D=     24  DATA=  0001  8800
     24         S/P=     -23  C/D=     -6  DATA=  0003  0000
     25         S/P=       3  C/D=     26  DATA=  0000  4400
     26         S/P=     -25  C/D=     27  DATA=  0000  0000
     27         S/P=      29  C/D=     28  DATA=  0000  0000
     28         S/P=     -27  C/D=     -7  DATA=  0000  0000
     29         S/P=     -26  C/D=     30  DATA=  0007  0000
     30         S/P=     -29  C/D=     -8  DATA=  0007  0000
     31         S/P=      -1  C/D=     32  DATA=  0007  FC00
     32         S/P=     -31  C/D=     33  DATA=  0007  5400
     33         S/P=     -32  C/D=     34  DATA=  0007  0000
     34         S/P=     -33  C/D=     35  DATA=  0000  0000
     35         S/P=     -34  C/D=     -9  DATA=  0000  0000
     36         S/P=      31  C/D=     37  DATA=  0006  DC00
     37         S/P=     -36  C/D=     38  DATA=  0004  4400
     38         S/P=     -37  C/D=     39  DATA=  0004  0000
     39         S/P=     -38  C/D=     40  DATA=  0000  0000
     40         S/P=     -39  C/D=    -10  DATA=  0000  0000
     41         S/P=      36  C/D=     42  DATA=  0005  FC00
     42         S/P=     -41  C/D=     43  DATA=  0001  5400
     43         S/P=     -42  C/D=     44  DATA=  0001  0000
     44         S/P=     -43  C/D=     45  DATA=  0000  0000
     45         S/P=     -44  C/D=    -11  DATA=  0000  0000
     46         S/P=      47  C/D=      0  DATA=  0000  0000
     47         S/P=      48  C/D=      0  DATA=  0000  0000
     48         S/P=      49  C/D=      0  DATA=  0000  0000
     49         S/P=       7  C/D=      0  DATA=  0000  0000
     50         S/P=       0  C/D=      0  DATA=  0000  0000

  DEBUG COMMAND = STOP
  STOPPING
  END OF DEBUG.
 ADAM COMMAND= DEL
 INPUT REG IND  >3
 ADAM COMMAND= GET
 INPUT REG IND  >1
 RETRIEVED DATA POINTS.
      5      .46091     .27927     .26559

 ADAM COMMAND= GET
 INPUT REG IND  >2
 NO DATA POINTS IN THE REGION.
 ADAM COMMAND= GET
 INPUT REG IND  >3
 NO DATA POINTS IN THE REGION.
 ADAM COMMAND= GET
 INPUT REG IND  >0
```

```
RETRIEVED DATA POINTS.
        7       .02341     .02927     .01559
        8       .11716     .12302     .10934
        5       .46091     .27927     .26559
       11       .89841     .02927     .51559
       10       .02341     .52927     .89059
        9       .89841     .90427     .89059

ADAM COMMAND= DEB
 DEBUG COMMAND = DUMP
 DUMP NODE CONTENTS.
   INDEX OF STARTING NODE = 0
   INDEX OF LAST NODE = 50
DUMP OF ADAM MAP.
    FROM       0      TO     50
       0      S/P=       20  C/D=       50  DATA=  0032   0000
       1      S/P=       -1  C/D=        2  DATA=  0000   FC00
       2      S/P=       41  C/D=       25  DATA=  0000   DC00
       3      S/P=       -2  C/D=       12  DATA=  0007   CC00
       4      S/P=        5  C/D=        0  DATA=  0000   0000
       5      S/P=        6  C/D=        0  DATA=  0000   0000
       6      S/P=       13  C/D=        0  DATA=  0000   0000
       7      S/P=        8  C/D=        0  DATA=  0000   0000
       8      S/P=        9  C/D=        0  DATA=  0000   0000
       9      S/P=       10  C/D=        0  DATA=  0000   0000
      10      S/P=       11  C/D=        0  DATA=  0000   0000
      11      S/P=       15  C/D=        0  DATA=  0000   0000
      12      S/P=       -3  C/D=       18  DATA=  0001   8C00
      13      S/P=       14  C/D=        0  DATA=  0000   0000
      14      S/P=       46  C/D=        0  DATA=  0000   0000
      15      S/P=       16  C/D=        0  DATA=  0000   0000
      16      S/P=       17  C/D=        0  DATA=  0000   0000
      17      S/P=       50  C/D=        0  DATA=  0000   0000
      18      S/P=      -12  C/D=       19  DATA=  0001   0C00
      19      S/P=      -18  C/D=       -5  DATA=  0000   0400
      20      S/P=       21  C/D=        0  DATA=  0000   0000
      21      S/P=       22  C/D=        0  DATA=  0000   0000
      22      S/P=       23  C/D=        0  DATA=  0000   0000
      23      S/P=       24  C/D=        0  DATA=  0000   0000
      24      S/P=        4  C/D=        0  DATA=  0000   0000
      25      S/P=        3  C/D=       26  DATA=  0000   4400
      26      S/P=      -25  C/D=       27  DATA=  0000   0000
      27      S/P=       29  C/D=       28  DATA=  0000   0000
      28      S/P=      -27  C/D=       -7  DATA=  0000   0000
      29      S/P=      -26  C/D=       30  DATA=  0007   0000
      30      S/P=      -29  C/D=       -8  DATA=  0007   0000
      31      S/P=       -1  C/D=       32  DATA=  0007   FC00
      32      S/P=      -31  C/D=       33  DATA=  0007   5400
      33      S/P=      -32  C/D=       34  DATA=  0007   0000
      34      S/P=      -33  C/D=       35  DATA=  0000   0000
      35      S/P=      -34  C/D=       -9  DATA=  0000   0000
      36      S/P=       31  C/D=       37  DATA=  0006   DC00
      37      S/P=      -36  C/D=       38  DATA=  0004   4400
      38      S/P=      -37  C/D=       39  DATA=  0004   0000
```

```
          39       S/P=    -38  C/D=      40  DATA=  0000   0000
          40       S/P=    -39  C/D=     -10  DATA=  0000   0000
          41       S/P=     36  C/D=      42  DATA=  0005   FC00
          42       S/P=    -41  C/D=      43  DATA=  0001   5400
          43       S/P=    -42  C/D=      44  DATA=  0001   0000
          44       S/P=    -43  C/D=      45  DATA=  0000   0000
          45       S/P=    -44  C/D=     -11  DATA=  0000   0000
          46       S/P=     47  C/D=       0  DATA=  0000   0000
          47       S/P=     48  C/D=       0  DATA=  0000   0000
          48       S/P=     49  C/D=       0  DATA=  0000   0000
          49       S/P=      7  C/D=       0  DATA=  0000   0000
          50       S/P=      0  C/D=       0  DATA=  0000   0000

  DEBUG COMMAND = STOP
  STOPPING
  END OF DEBUG.
 ADAM COMMAND= STOP
 STOP REQUESTED.
 END OF ADAM TEST.

PROGRAM TERMINATED AT #8333
STACK USED =  4653 OF  8128  HEAP USED =   882 OF 12302
LDOS READY
RESET *DO
```

Vita

James R. Holten III was born cn 18 April, 1949 in Paso Robles, California, to Mr. and Mrs. James R. Holten jr. He graduated from Illinois Valley High School, Cave Junction, Oregon, in 1967. In 1973 he graduated from Oregon State University with a Bachelor of Science in Mathematics and a Bachelor of Science in Computer Science. After graduation he enlisted in the Air Force, and in 1975 was admitted into Officer Training School. After commissioning on 16 July, 1975, he spent six years as a Missle Warning Programming Officer on phased array warning sites at Eglin Air Force Base, Florida; Otis Air Force Base, Massachusetts; and Beale Air Force Base, California. During this time he maintained computer programs for communications, radar function control, real time operating systems, and automated fault detection and isolation. He is married to the former Raymona A. Clinkingbeard of Ft. Walton Beach, Florida, and they have six children, Erin, Donald, James, Aghavni, Arlene, and Roger.

Permanent Address: 5839 Westside Rd.
Cave Junction, Oregon 97523

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/GCS/EE/82D-19	2. GOVT ACCESSION NO. AD-A124 674	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ASSOCIATIVE DATA ACCESS METHOD (ADAM)		5. TYPE OF REPORT & PERIOD COVERED MS Thesis
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) James R. Holten III		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Air Force Institute of Technology (AFIT-EN) Wright-Patterson AFB, Ohio 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE December, 1982
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Approved for public release: IAW AFR 190-17.
[illegible]
LYNN E. WOLAVER
Dean for Research and Professional Development
Air Force Institute of Technology (ATC)
Wright-Patterson AFB OH 45433

19 JAN [illegible]

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Associative Data Access
Computer Data Structures
Data Base Access
Associative Memory

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A software solution to the multikey access problem is presented. The result, ADAM, models associative memory techniques to obtain fast retrieval times and efficient data storage. A multidimensional tree structure is used. Each data item key is one dimension, and at each lower level in the tree each cimension is divided into successively smaller half-intervals. Unlike m-way trees with fixed sized nodes and K-D tree levels, each ADAM map level is a linear linked list. Each node of the ADAM level linear is the root of a subtree, or is the terminal node of a data item in the data set. The resulting data structure is, in many cases, more storage efficient than normal linear storage of the data items. This is due to the suppression of duplicate high order bits among the data items. The method allows retrieval of associative data subsets from the associative data set much faster than other multikey access techniques. Analysis of variations on ADAM are suggested, especially for application to very large (over 100000 data items per data set) multiuser databases.

END